21世纪高等学校规划教材 | 计算机科学与技术

分布式对象技术及其应用（第2版）

孟宪福　编著

清华大学出版社

分布式对象技术及其应用（第2版）

孟宪福　编著

清华大学出版社
北京

内容简介

分布式对象技术是在面向对象技术的基础上发展起来的，它要解决的主要问题是位于不同进程中的对象之间的调用问题。在中间件系统、Web 服务以及 SOA 等需要多程序协作的许多领域，分布式对象技术都发挥着重要作用。本书分 8 章，按照循序渐进的原则，从理论到实践逐步介绍分布式对象技术的典型代表 CORBA 和 Java RMI 的基本概念与程序设计规则。特别是，为了使读者能够尽快运用分布式对象技术来解决实际问题，在本书的最后两章完整地给出了基于 CORBA 和 Java RMI 的多个应用实例及其程序开发过程。

本书是作者根据多年的教学经验和实践体会编写而成的，在内容编排上尽量体现易学的特点，在文字叙述上力求条理清晰、简洁、便于读者阅读。

本书可以作为大专院校计算机专业研究生或高年级本科生的教材，也可以作为非计算机专业学生或软件开发人员的参考书或自学用书。

图书在版编目(CIP)数据

分布式对象技术及其应用/孟宪福编著. —2 版. —北京：清华大学出版社，2015
21 世纪高等学校规划教材 · 计算机科学与技术
ISBN 978-7-302-38596-7

Ⅰ. ①分… Ⅱ. ①孟… Ⅲ. ①分布式计算机系统－高等学校－教材 Ⅳ. ①TP338.8

中国版本图书馆 CIP 数据核字(2014)第 274530 号

责任编辑：梁　颖　薛　阳
封面设计：常雪影
责任校对：白　蕾
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课 件 下 载：http://www.tup.com.cn，010-62795954

印 刷 者：北京富博印刷有限公司
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：12.75　　**字　　数**：309 千字
版　　次：2008 年 11 月第 1 版　2015 年 1 月第 2 版　　**印　　次**：2015 年 1 月第1 次印刷
印　　数：1～2000
定　　价：35.00 元

产品编号：062071-01

出版说明

随着我国改革开放的进一步深化，高等教育也得到了快速发展，各地高校紧密结合地方经济建设发展需要，科学运用市场调节机制，加大了使用信息科学等现代科学技术提升、改造传统学科专业的投入力度，通过教育改革合理调整和配置了教育资源，优化了传统学科专业，积极为地方经济建设输送人才，为我国经济社会的快速、健康和可持续发展以及高等教育自身的改革发展做出了巨大贡献。但是，高等教育质量还需要进一步提高以适应经济社会发展的需要，不少高校的专业设置和结构不尽合理，教师队伍整体素质亟待提高，人才培养模式、教学内容和方法需要进一步转变，学生的实践能力和创新精神亟待加强。

教育部一直十分重视高等教育质量工作。2007 年 1 月，教育部下发了《关于实施高等学校本科教学质量与教学改革工程的意见》，计划实施“高等学校本科教学质量与教学改革工程”（简称“质量工程”），通过专业结构调整、课程教材建设、实践教学改革、教学团队建设等多项内容，进一步深化高等学校教学改革，提高人才培养的能力和水平，更好地满足经济社会发展对高素质人才的需要。在贯彻和落实教育部“质量工程”的过程中，各地高校发挥师资力量强、办学经验丰富、教学资源充裕等优势，对其特色专业及特色课程（群）加以规划、整理和总结，更新教学内容、改革课程体系，建设了一大批内容新、体系新、方法新、手段新的特色课程。在此基础上，经教育部相关教学指导委员会专家的指导和建议，清华大学出版社在多个领域精选各高校的特色课程，分别规划出版系列教材，以配合“质量工程”的实施，满足各高校教学质量和教学改革的需要。

为了深入贯彻落实教育部《关于加强高等学校本科教学工作，提高教学质量的若干意见》精神，紧密配合教育部已经启动的“高等学校教学质量与教学改革工程精品课程建设工作”，在有关专家、教授的倡议和有关部门的大力支持下，我们组织并成立了“清华大学出版社教材编审委员会”（以下简称“编委会”），旨在配合教育部制定精品课程教材的出版规划，讨论并实施精品课程教材的编写与出版工作。“编委会”成员皆来自全国各类高等学校教学与科研第一线的骨干教师，其中许多教师为各校相关院、系主管教学的院长或系主任。

按照教育部的要求，“编委会”一致认为，精品课程的建设工作从开始就要坚持高标准、严要求，处于一个比较高的起点上。精品课程教材应该能够反映各高校教学改革与课程建设的需要，要有特色风格、有创新性（新体系、新内容、新手段、新思路，教材的内容体系有较高的科学创新、技术创新和理念创新的含量）、先进性（对原有的学科体系有实质性的改革和发展，顺应并符合 21 世纪教学发展的规律，代表并引领课程发展的趋势和方向）、示范性（教材所体现的课程体系具有较广泛的辐射性和示范性）和一定的前瞻性。教材由个人申报或各校推荐（通过所在高校的“编委会”成员推荐），经“编委会”认真评审，最后由清华大学出版

社审定出版。

目前，针对计算机类和电子信息类相关专业成立了两个“编委会”，即“清华大学出版社计算机教材编审委员会”和“清华大学出版社电子信息教材编审委员会”。推出的特色精品教材包括：

（1）21世纪高等学校规划教材·计算机应用——高等学校各类专业，特别是非计算机专业的计算机应用类教材。

（2）21世纪高等学校规划教材·计算机科学与技术——高等学校计算机相关专业的教材。

（3）21世纪高等学校规划教材·电子信息——高等学校电子信息相关专业的教材。

（4）21世纪高等学校规划教材·软件工程——高等学校软件工程相关专业的教材。

（5）21世纪高等学校规划教材·信息管理与信息系统。

（6）21世纪高等学校规划教材·财经管理与应用。

（7）21世纪高等学校规划教材·电子商务。

（8）21世纪高等学校规划教材·物联网。

清华大学出版社经过三十多年的努力，在教材尤其是计算机和电子信息类专业教材出版方面树立了权威品牌，为我国的高等教育事业做出了重要贡献。清华版教材形成了技术准确、内容严谨的独特风格，这种风格将延续并反映在特色精品教材的建设中。

清华大学出版社教材编审委员会

联系人：魏江江

E-mail：weijj@tup.tsinghua.edu.cn

再版前言

随着软件应用领域的不断拓展，对程序设计技术的需求也越来越高，进而出现了分布式对象技术。分布式对象技术是在面向对象技术的基础上发展起来的，它要解决的主要问题是位于不同进程中的对象之间的调用问题。在中间件系统、Web 服务以及 SOA（Service Oriented Architecture：面向服务架构）的研究与开发等许多重要领域，分布式对象技术都发挥着不可替代的作用。本书共由 8 章组成，按照循序渐进的原则，从理论到实践逐步介绍分布式对象技术的典型代表——CORBA 的基本概念与程序设计规则，主要包括 CORBA 的组成与处理流程、IDL 接口定义语言、CORBA 客户端程序设计与服务器程序设计以及动态接口等内容。特别是，为了使读者能够尽快运用分布式对象技术来解决实际问题，本书利用两章的篇幅详细地介绍了基于 CORBA 的程序设计实例和基于 Java RMI 的程序设计实例，所给出的几个例子都是具有代表性的并具有实用价值的，通过对这些实例的学习，能够使读者进一步掌握分布式对象程序设计的要点，并能达到举一反三的目的。所给出的程序都是在实际的环境下调试完成的，以使读者能够尽快掌握分布式对象这门应用技术。

本书详细地介绍了分布式对象的基本内容，从理论到实践完整而系统地介绍了 CORBA 分布式对象系统设计规范和程序设计过程。本书的前 6 章主要是从理论的角度来介绍分布式对象系统的设计过程，后 2 章则从应用的角度来介绍分布式对象系统的实现过程。第 1 章简要介绍 Java 语言的基本内容，这是为阅读后续章节中的程序做准备的，所介绍的内容也仅局限在其他章节中需要使用的基本内容，包括基本语句、类的定义和接口等；第 2 章介绍分布式对象的基本概念以及 CORBA 的基本构成和处理过程，这一章的内容将为准确理解后续章节的内容打下基础；第 3 章主要介绍分布式对象系统的开发流程、IDL 语言以及从 IDL 到 Java 语言的映射，IDL 语言是基于 CORBA 的程序设计基础，而语言映射则为实现客户端和服务器提供了必需代码，这些代码包括 Stub 类、Skeleton 类、Holder 类和 Helper 类；第 4 章介绍 CORBA 客户端程序设计过程，主要包括 ORB 的初始化、分布式对象引用的获取以及分布式方法的调用等；第 5 章介绍 CORBA 服务器程序设计过程，主要包括 BOA 与 POA 的基本内容、分布式对象实现以及服务器进程实现等；第 6 章介绍动态接口技术，主要内容包括 any 类型的处理、接口仓库、动态启动接口 DII 和动态骨架接口 DSI 等；第 7 章介绍几个典型的 CORBA 实例，通过对这些实例的学习，能够准确了解 CORBA 应用系统的完整实现过程；第 8 章介绍 Java RMI 远程对象技术，详细说明了基于回调技术的分布式对象系统设计过程。

作者认为，分布式对象作为一门应用技术，要想学好它，除了掌握基本理论之外，还必须要加强实践环节。读者可以边学习边上机，刚开始时可以在给定的环境下调试本书中的例题，待学习一段时间之后，就可以调试自己编写的程序了。只有这样，才能加快学习进度，提高学习效率，真正掌握这门应用技术。

本书的出版得到了大连理工大学研究生院教改基金经费资助。

由于作者水平有限，经验不足，书中一定有不少缺点和错误，敬请有关老师、计算机工作者和广大读者批评指正。

作　者

2014 年 10 月

目 录

第1章 Java语言基础

Java语言是目前应用最广泛的面向对象程序设计语言之一，它具有面向对象、与平台无关、安全、稳定和多线程等优良特性。Java语言不仅可以用来开发大型的应用程序，而且特别适合于包括Internet应用等网络程序的开发。由于本书是以Java语言为基础来描述分布式对象技术的，因此，本章将对后续章节中需要使用的Java语言的基本内容进行简单的介绍。

1.1 Java语言的特点及其程序开发过程

Java语言的魅力主要体现在以下三个方面：

(1) 不管使用何种机器环境，只要有Java运行环境，Java的程序就可以执行。

(2) Java是一种拥有图形用户接口(GUI)和图像处理能力的新型的面向对象程序设计语言。

(3) Java语言程序可以作为Web页面的一部分来使用，这不仅体现在能使Web页面具有动态性的特点，而且体现在能够将Java语言程序从一台机器上快速下载到另一台机器上并运行这一强有力的功能上。

下面从程序设计语言方面来介绍Java语言的特点，同时简要介绍一下Java语言程序的开发过程。

1.1.1 Java语言的特点

从程序设计语言的角度来看，Java语言主要有如下一些主要特点。

1. 与C++语言相似

Java语言是不具有C++语言中的结构体、联合(共用体)、指针、预处理器等功能的非常简单的程序设计语言，如果熟悉C++语言的话，就可以比较容易地学会Java语言。

2. 以类为基础的面向对象程序设计语言

Java是以面向对象为基础而设计的语言，具有面向对象程序设计语言所具有的一切特点，如模块化(抽象)、数据隐藏、继承和多态等。同时，与C++语言不同的是，Java语言中的方法(包括main()方法)和全局变量等都只能作为类的成员来定义，而不能定义在类的外

部，这更体现出以类为基础的面向对象程序设计语言的特点。

3. 强类型检查功能

程序中所使用的字符串和数组等变量必须在使用之前进行类型定义。在编译时进行类型检查，以便在运行时将可能出现的问题降到最低。

4. 解释执行

Java 程序被编译为不依赖于任何机器的二进制形式的命令集，即字节码，其执行过程则是利用解释器来完成的。

5. 提供了丰富的类库

除了基本类库之外，Java 语言还提供了开发应用程序所不可缺少的 GUI 和网络功能等大量类库。

6. 可并发执行的语言

利用 Java 语言可以开发出同时由多个任务执行的多任务系统。通过利用多线程，可以实现并行处理。

7. 重视鲁棒性与安全性

Java 语言中没有指针类型，在编译时进行强类型检查。除此之外，在执行时还由解释器来检查字节码的正确性，其中，对字节码是否访问受限制的内存区域进行检查，从而能够保证系统运行的安全性和稳定性。

8. 性能

Java 程序的执行速度对于一般的实用程序来讲是足够快的，但一般认为比 C 语言慢，这主要是由于解释执行的原因。如果不对字节码进行解释执行，而是将其翻译为机器语言，那么就可以实现与 C 语言相同的执行速度。

9. 内存管理

程序设计者不需要明确地申请内存和释放内存，对内存的管理是自动进行的，使用完了的内存将被自动释放掉。

10. 与其他语言程序的接口

利用 C 语言等其他语言编写的程序可以作为动态装入库在 Java 程序中进行使用。Java 程序的一部分（即方法）可以利用其他语言来编写，这样的方法被称为 Native 方法。

1.1.2 Java 程序的开发过程

Java 语言程序的开发过程是由如下几部分组成的：

(1) 编辑 Java 源文件，并将其保存在以 java 为扩展名的文件中。

(2) 编译 Java 源文件,以生成扩展名为.class 的字节码文件。Java 编译器程序是 javac.exe。

(3) 运行 Java 程序。Java 程序可分为两类:一类是 Java 应用程序(即 Application),一类是 Java 小程序(即 Applet)。Java 应用程序一般需要通过 Java 解释器(java.exe)来解释执行其字节码文件;Java 小程序则必须通过支持 Java 标准的浏览器来解释执行。

下面的程序用于说明一个简单的 Java 应用程序的开发及运行过程。

```
//HelloJava.java
public class HelloJava
{
    public static void main(String args[ ])
    {
        System.out.println("Hello Java!");
    }
}
```

说明:

(1) 一个 Java 源程序是由若干个类组成的。在该程序中只有一个类,其类名为 HelloJava。

(2) 在一个 Java 应用程序中,必须有且只能有一个类含有 main()方法,包含 main()方法的类被称为主类,程序是从 main()方法开始执行的。

(3) 假设该程序被保存在 E:\Hello 目录下,则其编译过程如下:

```
E:\Hello\>javac HelloJava.java
```

编译完成后将生成一个 HelloJava.class 字节码文件,该字节码文件也被保存在 E:\Hello 目录下。

(4) 经编译之后,该程序的运行过程如下:

```
E:\Hello\>java HelloJava
```

屏幕上将显示如下信息:

```
Hello Java!
```

1.2 数据类型、变量、运算符和基本语句

Java 语言程序中所用到的每一个常量、变量及其函数等都是程序的基本操作对象,它们都隐式地或显式地与一种数据类型相联系,每种数据类型都表示了它的可能取值范围以及能在其上所进行的运算。Java 语言中提供了丰富的数据类型和运算符,运用这些数据类型和运算符能够进行复杂的数学运算。另外,由于程序的执行流程是通过语句来控制的,为此 Java 语言也提供了完善的语句支持。

本节主要讨论 Java 语言中的一些基本概念,如基本数据类型、运算符以及利用这些运算符来构成相应表达式的一些规则等,同时对基本语句也进行详细的介绍。

1.2.1 数据类型

Java 语言中的变量必须被定义为某种数据类型。Java 语言中的数据类型可以分为基本数据类型和引用数据类型两种。基本数据类型包括整型、浮点型和字符类型等；引用数据类型则包括对象和数组等。本节主要介绍基本数据类型。

Java 语言中的基本数据类型及其所占位数等信息如表 1.1 所示。

表 1.1 基本数据类型

基本数据类型	值的种类	位数	默认值
boolean	真、假	1	false
byte	带符号整数	8	0
short	带符号整数	16	0
int	带符号整数	32	0
long	带符号整数	64	0L
char	字符(Unicode)	16	'\u0000'
float	单精度浮点数	32	0.0f
double	双精度浮点数	64	0.0

在表示 float 型数据时，可以在数据的末尾加上 f 或 F，以区别于 double 类型数据。

以类的实例作为其值的变量被称为引用数据类型的变量。对于基本数据类型的变量来讲，其存储区域直接保存着值，而对于引用类型的变量来讲，其存储区域则保存着实际数据的引用。所谓引用，实际上在 C/C++语言中被称为指针，由于 Java 语言没有指针的概念，因此称其为引用。

类、接口和数组都属于引用类型。当定义了新的类或接口时，也就创建了新的引用类型。有关引用类型的基本内容将在后面详细介绍。

1.2.2 变量与常量

1. 变量

变量是指在程序运行过程中其值可以被改变的量。变量被区分为不同的类型，不同类型的变量在内存中占用不同的存储单元，以便用来保存相应变量的值。

变量的定义形式如下：

```
[访问修饰符][域修饰符] 数据类型名 变量名 [ = 初值];
```

如果在变量名的后面加上＝和初值，则在定义变量的同时为其初始化。如果没有指定初值，则对于基本数据类型的变量来讲，将被初始化为默认初值，而对于引用类型的变量来讲，将被初始化为 null。

Java 语言中的变量名(包括一般的标识符，如方法名、数组名和类型名等)是由字母、下划线、美元符号($)和数字组成的，并且第一个字符不能是数字。

作为习惯用法，变量名的首字符一般为小写字母，类名的首字符一般为大写字母。

访问修饰符包括 public、protected 和 private 三个；域修饰符包括 static、final、transient 和 volatile 四个，根据变量的意义可以同时指定多个。有关修饰符的具体内容将在后面详细介绍。实际上，修饰符主要用于指定变量的种类，而在 Java 语言中，可将变量分为如下几种类型。

1）成员变量

在类的定义中，在方法定义的外面作为类的成员定义的变量被称为成员变量。在定义成员变量时，根据是否加有 static 关键字，可将成员变量划分为类变量和实例变量。带有 static 关键字定义的变量被称为类变量，它是在类被装入时创建的，不管该类生成了多少个对象，该类变量也只有一个。类变量主要用于设定对所有对象都共享的数据；不带有 static 关键字定义的变量被称为实例变量，实例变量是在创建实例（对象）时生成的，每个实例都拥有不同的实例变量存储区域。

2）局部变量

在方法中或在方法中由左右花括号括起来的程序里面定义的变量为局部变量。局部变量仅在方法或左右花括号里面有效，方法执行结束或左右花括号内的程序执行结束时局部变量也就被释放了。由于局部变量不能被自动初始化，因此在程序中需要进行显式初始化。

由于局部变量仅在方法中或左右花括号中有效，同时仅在方法的执行期间存在，因此指定修饰符没有意义（如果指定的话编译出错）。但是，可以指定 final 修饰符。

3）方法的参数

方法的参数仅在方法（包括构造方法）的内部有效，方法执行结束之后就被释放了。对于基本类型的参数来讲，在方法被调用时是将参数的值传递给方法，而对于引用类型的参数来讲，则是将对象的引用传递给方法。

4）异常处理参数

被传递给异常处理的参数在异常处理结束后被释放。

2. 常量

常量是指在程序运行过程中其值不能被改变的量。在 Java 语言中，常量包括整型常量、浮点型常量、真假值、字符型常量、字符串常量和 null 等。

整型常量可有三种表示方式：一是十进制表示法，如 123，其每个数字位可以是 0～9，长整型常量可在末尾处加上 L 或 l；二是十六进制表示法，如果整型常量是以 0x 或 0X 开头的，那么就是十六进制表示的整型常量，如 0x8F，其每个数字位可以是 0～9，A～F（或 a～f）；三是八进制表示法，如果整型常量的最高位是 0，那么它就是以八进制形式表示的整型常量，如 0200，其每个数字位可以是 0～7。

浮点型常量有两种表示形式：一是十进制数形式，它是由数字和小数点来表示的。例如，3.141 59、－7.2 和 9.8 等。二是指数法形式，指数法又被称为科学记数法，它是由尾数、指数及字母 e（或 E）来表示的。例如：十进制数 180 000.0，用指数法可表示为 1.8e5。为了区分是 float 型常量还是 double 型常量，可在常量的末尾加上 F（f）或 D（d），如 1.23F 和 5.67D 等。

作为逻辑运算结果的真假值，是由 true 和 false 来表示的。

字符常量是由一对单引号括起来的一个字符，如'a'等。另外，在 Java 语言中还允许使

用一些特殊形式的字符型常量，这些字符型常量都是以反斜线字符开头的字符序列。例如：

- \n：为换行字符；
- \r：为回车字符；
- \b：为退格字符；
- \t：为制表字符，又被称为横向跳格字符；
- \'：为单引号字符；
- \"：为双引号字符；
- \\：为反斜线字符；
- \f：为换页字符；
- \…：八进制表示的 Unicode 字符，其中"…"为八进制数。

字符串常量是由双引号括起来的一串字符，如"Nothing"等。

null 表示不具有任何值的特殊常量。

1.2.3 运算符

在 Java 语言中，运算符被分为算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符和条件运算符等。

1. 算术运算符

算术运算符包括加(+)、减(-)、乘(*)、除(/)、求余(%)、增 1(++)和减 1(--)等。

2. 关系运算符

关系运算符包括大于(>)、大于等于(>=)、小于(<)、小于等于(<=)、等于(==)和不等于(!=)等。

3. 逻辑运算符

逻辑运算符包括逻辑与(&&)、逻辑或(||)和逻辑非(!)等。

4. 位运算符

位运算符包括算术左移(<<)、算术右移(>>)、逻辑右移(>>>)、按位与(&)、按位或(|)、按位异或(^)和按位取反(~)等。

5. 赋值运算符

赋值运算符是等号(=)，同时 Java 语言还允许在等号之前加上其他运算符以构成复合赋值运算符。例如，+=、%=、&=、^=和<<=等。

6. 条件运算符

条件运算符是?:，它是 Java 语言中唯一一个三元运算符。

7. new 运算符

在生成类的实例(对象)时,或者为数组申请内存空间时需要使用 new 运算符。new 运算符的具体用法将在后面具体介绍。

8. instanceof 运算符

instanceof 运算符用于判断所给定的对象是不是某个类的实例,若是返回 true,否则返回 false。例如:

```
if(p instanceof Exam == true)
{
   ⋮
}
```

用于判断对象 p 是不是 Exam 类的实例,若是,其返回值为 true。

9. 类型变换

类型变换是通过(类型名)运算符来实现的,例如,float a=(float)x/y; 是将 x 强制变换为 float 类型后再进行运算。

1.2.4 运算符的优先级

在一个表达式中如果有多个运算符,则计算是有先后顺序的,这种计算的先后顺序被称为相应运算符的优先级。例如,在上面介绍的算术运算符中,*、/、%运算符的优先级高于+、-运算符的优先级。表 1.2 给出了 Java 语言中各运算符的优先级。

表 1.2 运算符的优先级

优先级	运 算 符	优先级	运 算 符
1	[ ]、()、.、op++、op--	9	&
2	++op、--op、+op、-op、~、!	10	^
3	new (数据类型)	11	|
4	*、/、%	12	&&
5	+、-	13	||
6	<<、>>、>>>	14	?:
7	<>、<=、>=、instanceof	15	=、+=、-=、*=、/=、%=、^=、
8	==、!=		&=、|=、<<=、>>=、>>>=

1.2.5 数组

在 Java 语言中,数组的声明形式如下:

[访问修饰符][域修饰符]数据类型[ ] 数组名;

或者

[访问修饰符][域修饰符]数据类型 数组名[ ];

例如：

```
float [ ] abc;
float ABC[ ];
```

需要注意的是，在声明数组时并不指定数组元素的个数。

1. 数组的生成

数组被声明之后是不能直接使用的，需要利用 new 运算符为数组元素申请空间，然后才能使用。例如：

```
float [ ] abc;
abc = new float[10];
```

也可以在声明数组的同时为其申请空间。例如：

```
float ABC[ ] = new float[10];
ABC[2] = 5;
```

在 Java 语言中没有多维数组的概念，但由于 Java 语言中的数组元素本身还可以是数组，因此，可以实现像多维数组一样的功能。例如：

```
float [ ][ ]ma;
ma = new float[3][4];
ma[1][2] = 10;
```

这里的 ma 是具有 3 个元素的数组，每个元素又是具有 4 个元素的数组。

由于数组的每个元素又可以是数组，因此，作为数组元素的数组的长度是可以不相同的。例如：

```
float [ ][ ]mb = new float[3][ ];
mb[0] = new float[2];
mb[1] = new float[3];
mb[2] = new float[5];
```

显然，这样定义的数组，每行的元素个数是不同的。

2. 数组元素的初始化

数组元素的初始化可采用如下两种方式：

```
数据类型名[ ] 数组名 = {初值 1,初值 2,…};
数据类型名 数组名[ ] = {初值 1,初值 2,…};
```

或者

```
数据类型名[ ] 数组名 = new 数据类型名[ ]{初值 1,初值 2,…};
数据类型名 数组名[ ] = new 数据类型名[ ]{初值 1,初值 2,…};
```

例如：

```
float mc[ ] = {11.1,22.2,33.3};
```

或者

```
float mc[ ] = new float[ ]{11.1,22.2,33.3};
```

又如,String 类型数组的初始化方式如下:

```
String sa[ ] = { "aaa","bbb","ccc"};
```

或者

```
String sa[ ] = new String[ ]{ "aaa","bbb","ccc"};
```

需要注意的是,对于数组元素本身又是数组的数组的初始化方式如下:

```
int ia[ ][ ] = {{11,22,33},{55,66,77}};
String sc[ ][ ] = {{ "aa","bb","cc"},{"dd","ee","ff"}};
```

这里,ia 和 sc 都是具有两个元素的数组,每个元素又都是具有三个元素的数组。

3. 数组元素的引用与数组的长度

与其他语言相同,数组元素是通过数组名和下标来引用的。在 Java 语言中,数组的下标是从 0 开始的。例如:

```
mc[0] = 10.9;
int x = mc[0] * 5;
ia[1][0] = 10;
```

数组的长度,亦即数组中元素的个数可以利用如下方式获取:

```
int a[ ] = new int[8];
int len = a.length;
```

这里,len 的值为 8。

1.2.6 字符串

在 Java 语言中提供了两个类用于对字符串进行处理:一是用于字符串常量的 String 类,另一个是可以改变字符串内容的 StringBuffer 类。

字符串常量是 String 类的实例,String 类的对象一旦生成之后,其内容就不能再改变了。与此不同,对 StringBuffer 类的对象而言,可以进行追加和插入等操作。

在 Java 程序中,只要是用双引号括起的文字就自动生成了 String 类的实例。例如:

```
"This String"
```

该字符串本身实际上就创建了 String 对象。

与其他类相同,也可以利用 new 运算符来创建 String 对象。由于 String 类中定义了多个构造函数,因此在创建对象时可以传递多种参数。例如:

```
char ch[ ] = { 'D','a','t','a'};
String sch = new String(ch);
```

为了生成 StringBuffer 类的实例,也需要利用 new 运算符。StringBuffer 类的构造函

数的参数可以是保存字符串的缓冲区的大小，也可以是字符串的初值。例如：

```
StringBuffer sbuff = new StringBuffer(10);
```

表示 sbuff 对象中可以保存的字符串的长度为 10。又如：

```
StringBuffer sp = new StringBuffer("Nothing");
```

表示创建初值为“Nothing”的 StringBuffer 对象 sp。

在对 StringBuffer 对象进行操作时，如果字符串的长度超过了 StringBuffer 对象的容量，则系统将自动扩大容量。

在 String 类中和 StringBuffer 类中提供了很多字符串操作方法，如字符串比较等，在此不一一介绍了。

1.2.7 基本数据类型包装类

1. 包装类的作用

在对数值进行处理时，往往会使用 int 或 float 这种基本类型的数据，但如果将字符串和基本类型的数值一起运算的话就会出现问题。为此，需要将字符串所表示的数值转换为 float 等类型之后再参加运算。在 Java 语言的 java.lang 包中提供了在 String 与基本数据类型之间进行类型变换的类，这些类被称为包装类（wrapped class），它们都是 Number 类的子类。这些包装类包括：

```
Boolean,Byte,Character,Integer,Short,Long,Float,Double
```

包装类的名字与基本数据类型的名字是相同的，只是第一个字母是大写的。由于这些包装类的值是基本数据类型的常量或变量，因此，可以利用 new 运算符来生成其对象。例如：

```
double d = 22.2;
Double dobj = new Double(d);
```

2. 包装类中的方法

包装类中的方法可以分为两类：一是检查对象的值，或者将其变换为基本数据类型的实例方法；二是检查参数的数据，或者将其变换为别的数据类型的类方法。

属于第一种类型的方法需要利用包装类的实例来调用。例如，下面的语句是将 Double 对象 dobj 的值变换为 float 类型：

```
float f = dobj.floatValue();
```

属于第二种类型的方法主要用于对所传递的参数进行处理。由于这些方法被定义为 static 方法，因此，只要利用类名就可以调用。例如，下面的语句用于判断 char 类型的 c 变量的值是否是小写字母：

```
char c = 'm';
boolean b = Character.isLowerCase(c);
```

包装类中定义了很多方法，在此不再一一叙述。

1.2.8 基本语句

1. if 语句

if 语句是条件选择语句，它通过对给定条件的判断来决定所要执行的操作。

if 语句的一般形式如下：

```
if(表达式)
    语句 1
else
    语句 2
```

if 语句的执行过程是：首先计算表达式的值，若表达式的值为“真”，则执行语句 1；若表达式的值为“假”，则执行语句 2。如果语句 1 或语句 2 是由多个语句组成的，则需要用左右花括号括起来。

if 语句中的表达式可以是关系表达式、逻辑表达式或算术表达式等。例如：

```
if(a<b)
   M=a;
else
   M=b;
```

2. switch 语句

上面介绍的 if 语句，一般适用于两路选择，即在两个分支中选择一个执行，尽管可以通过 if 语句的嵌套形式来实现多路选择的目的，但这样做的结果使得 if 语句的嵌套层次太多，降低了程序的可读性。Java 语言中的 switch 语句，提供了更方便地进行多路选择的功能。

switch 语句的一般形式如下：

```
switch(表达式)
{
    case  常量表达式 1:
                语句 1;
                break;
    case  常量表达式 2:
                语句 2;
                break;
            ⋮
    case  常量表达式 n:
                语句 n;
                break;
    default:
                语句 n+1;
                break;
}
```

其中 default 及其语句部分可以同时省略。

switch 语句的执行过程是：首先计算 switch 后面圆括号内表达式的值，若此值等于某个 case 后面的常量表达式的值，则转向该 case 后面的语句去执行；若表达式的值不等于任何 case 后面的常量表达式的值，则转向 default 后面的语句去执行，如果没有 default 部分，则将不执行 switch 语句中的任何语句，而直接转到 switch 语句后面的语句去执行。

在使用 switch 语句时，应注意如下几个问题：

(1) switch 后面圆括号内的表达式的值和 case 后面的常量表达式的值，都必须是整型的或字符型的，不允许是浮点型的。

(2) 同一个 switch 语句中的所有 case 后面的常量表达式的值都必须互不相同。例如：

```
switch(c)
{
    case '*':
         s++;
         break;
    case '*':
         s++;
         break;
}
```

是不合法的。

(3) switch 语句中的 case 和 default 的出现次序可以是任意的，也就是说，default 也可以位于 case 的前面，且 case 的次序也不要求按常量表达式的顺序排列。

(4) 由于 switch 语句中的“case 常量表达式”部分只起语句标号的作用，而不进行条件判断，所以，在执行完某个 case 后面的语句后，将自动转到该语句后面的语句去执行，直到遇到 switch 语句的右花括号或 break 语句为止，而不再进行条件判断。例如：

```
switch(n)
{
    case 1:
         x = 1;
    case 2:
         x = 2;
}
```

当 n=1 时，将连续执行下面两个语句：

```
x = 1;
x = 2;
```

所以，在执行完一个 case 分支后，一般应跳出 switch 语句，转到下一条语句执行，这样，可在一个 case 的结束后，下一个 case 开始前，插入一个 break 语句，一旦执行到 break 语句，将立即跳出 switch 语句。例如：

```
switch(n)
{
    case 1:
         x = 1;
```

```
            break;
        case 2:
            x = 2;
            break;
}
```

(5) 每个 case 的后面既可以是一个语句,也可以是多个语句,当是多个语句的时候,也不需要用花括号括起来。

(6) 多个 case 的后面可以共用一组执行语句。例如:

```
switch(n)
{
    case 1:
    case 2:
        x = 10;
        break;
    :
}
```

它表示当 n=1 或 n=2 时,都执行下列两个语句:

```
x = 10;
break;
```

下面的程序段给出了 switch 语句的基本用法。

```
switch(x)
{
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    default:
        System.out.println("Other days");
        break;
}
```

3. while 语句

while 语句的一般形式如下所示:

```
while (表达式)
    语句      (即循环体)
```

其执行过程是:先计算 while 后面圆括号内的表达式的值,如果其值为"真",则执行语句(又称循环体),然后再次计算 while 后面圆括号内表达式的值,并重复上述过程,直到表达式的值为"假"时,退出循环,并转入下一语句去执行。例如:

```
int a = 0,b = 1;
while(b<10)
```

```
{
    a += b;
    b++;
}
```

使用 while 语句时，需要注意如下几个问题：

（1）while 语句的特点是先判断表达式的值，然后执行循环体中的语句，因此，如果表达式的值一开始就为“假”，则循环体将一次也不执行。

（2）当循环体是由多个语句组成时，必须用左、右花括号括起来。例如：

```
while(x>0)
{
    s += x;
    x--;
}
```

（3）为了使循环最终能够结束，而不至于产生“死循环”，每执行一次循环体，表达式的值都应该有所变化，这既可以在表达式本身中实现，也可以在循环体中实现。

下面的程序段给出了 while 语句的基本用法：

```
int a = 10;
while(a>0)
{
    System.out.println(a);
    a--;
}
```

4. for 语句

for 循环语句的一般形式如下：

```
for(表达式 1;表达式 2;表达式 3)
        语句    （即循环体）
```

其执行过程是：首先求解表达式 1，然后求解表达式 2，若表达式 2 的值为“真”，则执行循环体中的语句，接着求解表达式 3，并再次求解表达式 2，若表达式 2 的值为“真”，再一次执行循环体，如此循环，直到表达式 2 的值为“假”时退出循环，并转到下一语句执行。由 for 语句的执行过程可知，for 语句中的表达式 1 仅在循环开始前执行一次，以后就不再执行了。

for 语句的功能可用 while 语句描述如下：

```
表达式 1;
while (表达式 2)
{
    语句  （即循环体）
    表达式 3;
}
```

使用 for 语句时，需要注意如下几个问题：

（1）for 语句中的任何一个表达式都可以省略，但其中的分号一定要保留，当省略表达式 2 时，相当于“无限循环”（循环条件总为“真”），这时就需要在 for 语句的循环体中设置相

应的语句来结束循环。

(2) 如果 for 语句的循环体部分是由多个语句组成的,则必须用左、右花括号括起来。

(3) 需要特别注意的是,for 语句的表达式 1、表达式 2 和表达式 3 之间必须要用分号间隔,而不能使用其他符号。

下面的程序段给出了 for 语句的基本用法:

```
int a = 2;
for(int i = 1;i<10;i++)
{
    System.out.println(a * i);
    a++;
}
```

5. do-while 语句

do-while 语句的一般形式如下:

```
do
   语句    (即循环体)
while (表达式);
```

其执行过程是:先执行循环体部分的语句,再计算 while 后面圆括号内表达式的值,若表达式的值为"真",则再次执行循环体,如此循环,直到表达式的值为"假"时,结束循环,并转到下一条语句执行。

使用 do-while 语句应注意如下几个问题:

(1) 由于 do-while 语句是先执行循环体,然后再判断表达式的值,所以,无论一开始表达式的值为"真"还是为"假",循环体中的语句都至少被执行一次,这一点同 while 语句是有区别的。

(2) 如果 do-while 语句的循环体是由多个语句组成的话,则必须用左、右花括号括起来。例如:

```
do
{
    s += x;
    x--;
} while(x>0);
```

下面的程序段给出了 do-while 语句的基本用法:

```
int a = 10;
do
{
    System.out.println(a);
    a--;
} while(a>0);
```

6. break 语句

break 语句能够跳出 switch 语句,而转入下一条语句执行,这在前面已经介绍过。实际

上，break 语句也能够终止循环语句的执行。

break 语句的一般形式如下所示：

```
break;
```

或者

```
break 标号;
```

其执行过程是：终止对 switch 语句或循环语句的执行，即跳出这两种语句，而转入下一条语句执行。

对于多重循环语句来讲，break 语句只能跳出它所在的那层循环，为了使 break 语句能够跳出所有循环语句，可以在 break 语句的后面指定标号。例如：

```
for(int i = 0;i<10;i++)
    for(int j = 0;j<10;j++)
    {
    if(x[i] == y[j])
        break equ;
    }
equ:
    ⋮
```

这里，当 x[i]==y[j]时，break 语句将跳出双重循环语句。如果 break 语句的后面没有标号的话，则只能跳出内层循环。

7. continue 语句

continue 语句只能出现在循环体中，它的作用与 break 语句有些相似，但它并不造成整个循环语句的终止，而是跳过循环体中位于 continue 语句后面的其他语句，并立即开始下一轮循环。

continue 语句的一般形式如下：

```
continue;
```

或者

```
continue 标号;
```

其执行过程是：终止当前这一轮循环，即跳过循环体中位于 continue 后面的语句而立即开始下一轮循环：对于 while 和 do-while 语句来讲，这意味着立即计算循环是否终止的表达式（即执行条件测试部分），而对于 for 语句来讲，这意味着立即计算表达式 3。

对于多重循环语句来讲，continue 语句只能开始它所在的那层循环的下一轮循环，为了使 continue 语句能够开始外层循环的下一轮循环，可以在 continue 语句的后面指定标号。例如：

```
equ:for(int i = 0;i<10;i++)
      for(int j = 0;j<10;j++)
      {
        if(x[i] == y[j])
```

```
            continue equ;
        }
```

这里，当 x[i]==y[j]时，continue 语句将终止当前这一轮循环，而转入外层的下一轮循环。如果 continue 语句的后面没有标号，则只能转入内层的下一轮循环。

1.3 类

类是 Java 语言的重要组成部分，实际上，Java 语言的程序就是类的定义和接口定义的集合。本节主要介绍与类有关的基本内容。

1.3.1 对象的生成与引用

对象是利用 new 运算符来生成的，new 运算符将返回所生成的类对象的引用。类对象的生成方式如下：

```
new 类名(传递给构造方法的参数)
```

在程序中一般是将由 new 运算符生成的对象保存在变量中，并利用变量来操作对象。例如，下面的语句是生成 java.awt 包中的 Rectangle 类的对象，并将其保存在 rect 变量中：

```
Rectangle rect = new Rectangle(0,0,10,20);
```

在利用 new 运算符来生成对象时，将自动调用具有相同参数个数和参数类型的构造方法。构造方法是其方法名与类名相同的特殊方法，将在下面进行详细介绍。

生成了对象之后，就可以利用对象来访问其实例成员变量和实例方法。例如：

```
rect.width = 100;
rect.move(10,10);
```

而对于类变量和类方法来讲，直接通过类名就可以访问。例如，圆周率的值是在 java.lang 包中的 Math 类中作为类变量定义的，其引用方法如下：

```
double y = Math.PI;
```

又如，生成随机数的 random()方法是在 Math 类中定义的类方法，其调用方法如下：

```
double rm = Math.random();
```

1.3.2 this 与 super

this 是表示当前对象的关键字，通常用于引用当前对象的成员变量或者将当前对象作为参数进行传递的情况。

下述写法表示引用本对象中的成员变量：

```
this.成员变量名
```

利用 this 关键字来引用自身成员变量的情况主要有两种：一是方法的参数与成员变量同名，二是局部变量与成员变量同名。在这两种情况下，由于局部变量优先的原则，如果直

接利用变量名的话将引用的是局部变量或方法的参数，为了引用成员变量就需要使用 this 关键字。例如：

```
class Test
{
    int cnt;
    void func1(int cnt)
    {
        this.cnt = cnt;  //将参数的 cnt 值赋给成员变量 cnt
        ⋮
    }
    void func2()
    {
        int cnt;
        cnt = this.cnt;  //将成员变量的 cnt 值赋给局部变量 cnt
        ⋮
    }
    ⋮
}
```

除此之外，当需要以当前对象为参数来调用方法时也可以使用 this 关键字。例如：

```
func(this);
```

super 关键字用于表示父类，当需要引用直接父类的成员时使用。其用法如下：

```
super.成员变量名
super.方法名(参数)
```

super 关键字主要用在父类中的成员（包括成员变量与方法）与子类中的成员同名且需要引用父类中的成员的情况。例如：

```
class Base
{
    int height = 10;
    void func()
    {
        ⋮
    }
}
class Derived extends Base
{
    int height = 20;
    void func()
    {
        System.out.println(height);          //显示 20
        System.out.println(super.height);    //显示 10
        super.func();                        //调用 Base 类中的 func()方法
    }
}
```

1.3.3 类的定义

Java 语言中的类的定义形式如下(方括号内的部分可以省略):

```
[类修饰符] class 类名 [extends 父类名] [implements 接口名 1[,接口名 2…]]
{
    成员变量的定义
    方法的定义
}
```

说明:

(1) class 是定义类的关键字。

(2) 类修饰符包括 public、abstract 和 final 等。

public:用于定义从该类所在的包(package)的外部也能使用的类。如果不加 public,则该类只能在其所在的包的内部使用。包含 public 类的源文件必须被保存在名为类名且扩展名为.java 的文件中。

abstract:用于定义抽象类。所谓抽象类就是不能生成实例的类,主要用于定义其子类。

final:用于定义不能生成子类的类。在类的层次上,final 类属于叶子的类,不能作为父类出现。

需要注意的是,在定义类时,abstract 和 final 不能同时出现。

(3) extends 关键字用于指定父类。在 Java 语言中,所有的类都自动继承 java.lang.Object 类,所以类的定义,实际上也就是子类的定义。由于 Java 语言只提供单一继承功能,因此,在 extends 的后面只能有一个类名。能够被继承的成员变量和方法的访问修饰符包括 public 和 protected。如果父类和子类在同一个包中,则父类中没有指定访问权限的成员变量和方法也能被子类继承。不能被继承的成员变量和方法的访问修饰符为 private,同时构造方法也不能被继承。

(4) implements 关键字用于指定需要实现的接口(interface)。如果需要实现多个接口的话,则接口名之间由逗号隔开。

1.3.4 成员变量的定义

成员变量的定义形式如下:

```
[访问修饰符][static][final][transient][volatile] 数据类型  变量名
```

说明:

(1) 最简单的成员变量定义方式可以只包括数据类型和变量名。如果指定修饰符的话,可以确定对该变量的访问控制权限、实例变量还是类变量以及常量等。

(2) 访问修饰符包括 public、private、protected 和无修饰符。

- public:用于定义从哪个类都可以访问的变量或方法。
- private:用于定义只能在本类的方法中才能访问的成员变量或方法。
- protected:被定义为 protected 的成员变量,既可以从本类的方法中访问,也可以从

同一个包的其他类中访问，同时，从子类中也可以访问（即使子类是在别的包中定义的也可以访问）。

- 无修饰符：从同一个包所定义的类中可以访问。

（3）static 用于定义类变量，如果不指定 static 则是定义实例变量。

（4）final 用于定义常量。例如：

```
public static final double PI = 3.141 59;
```

（5）transient 表示所定义的变量是临时的，而不是对象的固定部分。在对象序列化时，由该关键字所指定的变量其值不能被保存。

（6）volatile 表示所定义的变量的值可以并发地由其他线程来修正，而其值仍能保持连贯性。

1.3.5 方法的定义

方法的定义形式如下：

```
[访问修饰符][static][final][abstract][native][synchronized] 返回值类型 方法名([参数表])
                                                    [throws 异常类名表]
{
    方法体
}
```

说明：

（1）访问修饰符包括 public、protected、private 和无修饰符，其意义与上述相同。

（2）static 用于定义类方法。如果不指定 static 则是定义实例方法。

（3）final 用于定义在子类中不能被改写的方法。这同用 final 定义的变量其值不能被修改是一样的。

（4）abstract 用于定义抽象方法，即只定义方法的框架（包括返回值类型、方法名和参数类型等），而不定义方法体。与 final 方法相反，由 abstract 定义的方法必须要在子类中重新定义。

需要注意的是，在抽象类中可以定义非抽象方法，即有函数体的方法，但抽象类不能定义对象，同时抽象方法只能定义在抽象类中。由于 static 方法不能被子类继承，所以 static 方法不能被定义为 abstract。

下面是抽象方法定义的例子：

```
abstract float weight(person P);
```

（5）native 关键字用于说明方法的主体是用别的程序设计语言编写的。由于此方法的主体是在别的地方定义的，所以其说明方式如下：

```
public native double get(double r);
```

（6）synchronized 关键字用于定义同步方法。当某个程序是由多线程执行时，被定义为 synchronized 的方法同时只能由一个线程执行。当某个线程正在执行该方法时，其他线程要想执行该方法，就必须等到前一个线程执行结束。

(7) 方法的返回值是由 return 语句返回的,其形式如下:

```
return 表达式;
```

其中,"表达式"部分可以省略,表示不返回任何值,此时,方法的返回值类型应该指定为 void。

(8) "参数表"部分既可以是一个参数,也可以是由逗号隔开的多个参数,当然也可以没有参数。

(9) throws 用于说明在该方法中能够抛出的异常类型。

1.3.6 构造方法的定义与使用

构造方法是在定义对象时被自动调用的特殊方法,其名字与类名相同。构造方法主要用于在定义对象时为成员变量进行初始化。构造方法的定义方式如下:

```
类名(参数表)
{
    构造方法体
}
```

需要注意的是,构造方法没有返回值类型说明,这一点与其他方法的定义是不同的。

构造方法体的第一个语句可以是 this(参数表)或 super(参数表),分别表示调用当前对象的其他构造方法和调用父类的构造方法。括号里的参数个数及其类型决定了要调用哪个构造方法。如果构造方法体的第一个语句不是 this(参数表)或 super(参数表),则默认为 super(),即调用父类中无参数的构造方法。例如:

```
class BaseA
{
    double m;
    BaseA()
    {
        this(5.0);          //执行 BaseA(5.0)
    }
    BaseA(double m)
    {
        this.m = m;
    }
}
class DerivedA extends BaseA
{
    double a;
    DerivedA(double b)
    {
        a = b;              //执行 BaseA()
    }
    DerivedA(double x,double y)
    {
        super(x);           //执行 BaseA(x)
        a = y;
```

```
    }
}
```

需要注意的是，this(参数表)与 super(参数表)语句只能作为构造方法体的第一个语句使用。

1.3.7 static 块

static 块(static block)主要用于对类变量进行初始化。其基本用法如下：

```
static{ 类变量的初始化 }
```

例如：

```
class SBlock
{
    static int a[] = new int[10];
    int x = 5;
    static
    {
        for(int i = 0;i<10;i++)
          a[i] = i * 2;
    }
}
```

需要注意的是，在 static 块中，既不能使用实例变量，也不能使用 this 和 super 关键字。比如，如果在上述 static 块中有语句 a[i]=x; 则出错。

由于一般可执行语句只能写在方法体中，因此，static 块为类变量的初始化提供了方便。

1.3.8 对象的释放

在 Java 语言中，当某个对象不再需要时就可以释放掉。为了释放对象，Java 语言中的垃圾收集程序将定期和自动地进行处理。因此，程序设计人员不再需要编写对象释放处理程序。同时，哪些对象应该被释放是由垃圾收集程序自动判断的。

1.4 接口与异常处理

1.4.1 接口

接口用于定义没有方法体的方法(即抽象方法)和常量。在定义类时所使用的 implements 关键字就是用于指定接口的，以便在类中能够使用接口中定义的常量和实现接口中定义的方法。由此可见，从 interface 中不能继承变量，同时从 interface 中也不能继承方法体。

接口的定义方式如下：

```
[public] interface 接口名 [extends 接口名 1[,接口名 2…]]
```

```
{
    常量定义
    抽象方法定义
}
```

说明：

(1) public 修饰符用于定义在所有包中的所有类都能使用的接口，如果没有 public 修饰符，则所定义的接口的作用域仅局限于其所在的包。

(2) extends 用于指定父接口，与类的继承不同，可以指定多个接口，也就是可以从多个接口中继承常量和抽象方法定义。

(3) interface 中的常量定义，即使不指定修饰符也被看作是 public、static 和 final。这些修饰符不需要指定。在定义常量时必须要指定初值，同时，在 interface 定义中不能使用 static 块来初始化常量。

(4) interface 中定义的方法被看作是 public 和 abstract，这些修饰符不需要指定。同时，不能定义方法体。例如：

```
interface Readable
{
    int r = 50;
    void read(int m);
}
```

当某个类需要实现一个接口时，在定义类时需要利用 implements 来指定该接口。同时，在接口中定义的所有方法，都必须在类中给出其方法体的定义。由于 interface 中定义的方法都是 public，因此，在类中也必须指定为 public 修饰符。例如：

```
class ImpR implements Readable
{
    public void read(int m)
    {
        …        //给出方法体的定义
    }
}
```

需要注意的是，interface 也是一种引用数据类型，与基本数据类型以及其他引用类型一样，接口名也可以作为数据类型使用。例如：

```
class Exam
{
    public static void main(String args[ ])
    {
        Readable r = new ImpR();        //生成 ImpR 对象
        r.read(8);                      //调用 ImpR 类中定义的方法
    }
}
```

1.4.2 异常处理

所谓异常处理，是指程序中出现错误时的处理。Java 语言中的异常处理结构如下：

```
try
{
    语句
}
catch(异常类名 1 变量名 1)
{
    语句        //异常处理
}
catch(异常类名 2 变量名 2)
{
    语句        //异常处理
}
finally
{
    语句        //后处理
}
```

try 部分的语句包含那些有可能抛出异常的语句及方法调用。catch 部分的语句用于描述当发生相应异常时的处理过程。finally 部分的语句无论异常是否发生，在整个处理过程结束之后都要执行，也就是当在 try 程序段中发生异常时，则终止 try 程序段的执行，转而执行相应的 catch 程序段中的异常处理程序，然后执行 finally 程序段，以进行相应的后处理；当 try 程序段的执行正常结束时，也转去执行 finally 程序段，以进行相应的后处理。finally 程序段主要用于对在 try 程序段中打开的文件进行关闭等处理。

在上述处理过程中，要想在 catch 部分能捕捉到异常，就必须在 try 部分所调用的方法中能抛出异常对象，这主要涉及 throws 和 throw 两个操作。例如，下面定义的方法能够抛出异常对象：

```
void myMethod()throws MyException
{
    ⋮
  throw new MyException;
    ⋮
}
```

由上述定义不难看出，当在方法体中需要利用 throw 语句抛出异常时，在方法的首部就必须利用 throws 来给出相应的异常类名，当有多个异常类时，类名与类名之间需用逗号分隔。例如：

```
void myMethod()throws MyException,HeException
{
    ⋮
    if(…)
      throw new MyException;      //调用默认构造方法创建异常类 MyException 的对象
    ⋮
    if(…)
      throw new HeException;      //调用默认构造方法创建异常类 HeException 的对象
    ⋮
}
```

在介绍了能够抛出异常对象的方法定义方式之后，还要介绍一下异常类的定义。

在Java语言中，异常类的定义与一般类的定义是有区别的，这主要体现在定义方式及其应该继承的类等方面。一般来讲，在定义异常类时需要继承Exception类。

例如，MyException异常类的定义如下：

```
class MyException extends Exception
{
    MyException(){ }
    MyException(String message)
    {
      super(message);
    }
}
```

在上述异常类的定义中，带有参数的构造函数所指定的参数，可以利用getMessage()方法来获取。这里，getMessage()方法是在Throwable类中定义的，而Throwable是Exception类的父类。例如：

```
void myMethod()
{
    ⋮
    try
    {
        //调用读文件方法
        ⋮
    }
    catch(IOException e1)
    {
        System.out.println(e1.getMessage());
    }
    catch(NumberFormatException e2)
    {
        System.out.println(e2.getMessage());
    }
}
```

显然，在利用带有参数的构造方法来生成异常对象并抛出时，应该给出参数信息。例如：

```
if(…)
  throw new MyException("Error occurred!");
```

需要注意的是，如果在一个方法中需要捕获由其他方法抛出的异常，则在本方法的定义首部，不需要利用throws语句来对该异常类名进行说明。

1.4.3 包

包(package)是由相关的类和接口所组成的类库。由Java语言提供的类被分类为多个类库也就是多个包来进行管理。包的作用就在于可以防止名字碰撞问题以及可以对类进行

分类管理。

包是按照层次结构来管理的，这一点同文件系统的目录结构相似。图 1.1 给出了 java 包的部分构成。在 java 包的下面有 lang、io 和 awt 等子包，同时，在 awt 包的下面还包含 image 子包和 Button 等类型声明。因此，每个包都包含子包和类型声明（即类和接口的定义）。

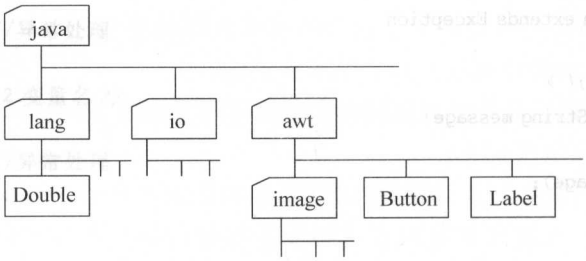

图 1.1 java 包的层次结构

根据包的层次构造，利用“.”符号将包名和子包名等连接起来就可以识别子包、类和接口的定义。例如，java.awt.Button 是识别 Button 类的名字（被称为完全限定名），同时该名字也与该类的字节码文件所保存的路径相对应。比如，pack1.pack2.Exam 类的字节码文件，在 Windows 系统下，将被保存在 pack1\pack2 目录下。

可以将相关的类和接口的集合定义为一个包。包的定义是利用 package 声明实现的。package 声明的形式如下：

```
package 包名;
```

定义了包名之后，包中所包含的字节码文件（.class 文件）将被保存在与包同名的目录下。例如：

```
package pack1.pack2;
```

该包中的所有字节码文件将被保存在 pack1 目录下的 pack2 子目录下。

如果 Java 程序中没有 package 声明，则所定义的类和接口将被保存在没有名字的默认包中。为了防止名字上的冲突，在程序中还是应该定义一个唯一的包名。

为了在程序中使用包，需要利用 import 声明，其格式如下：

```
import 包名.*;
import 包名.类名;
import 包名.接口名;
```

经过 import 声明之后，在程序中对类名或接口名的引用就不再需要包名了。import 声明中的 *，表示该包中的所有 public 类和接口都可以直接引用。

1.4.4 命令行参数

Java 应用程序的执行是从 main()方法开始的。main()方法有一个 String 类型的数组参数，该参数用于传递命令行参数。在应用程序中即使不利用该参数，也必须给出该参数的定义。

main()方法的定义形式如下：

```
public static void main(String args[ ])
{
    方法体
}
```

向 main()方法传递的参数，是在启动 java 解释器时，在类名的后面给出的。

下面的程序用于说明命令行参数的意义及用法。

```
class CommLine
{
    public static void main(String args[ ])
    {
        for(int i = 0;i<args.length;i++)
            System.out.println(args[i]);
    }
}
```

该程序的执行命令及其执行结果如下：

```
D:\>java CommLine Nothing is hard!
Nothing
is
hard!
```

需要注意的是，命令行上的参数是作为字符串传递给 main()方法的，如果在程序中希望以数值的方式使用，就需要将字符串转换为数值，这一工作可以利用前述的包装类中的方法来完成。例如，如果希望将命令行的第一个参数转换为 float 型数据后使用，可以利用下述语句：

```
float f = Float.valueOf(args[0]).floatValue();
```

该语句首先利用 Float 类中的 valueOf()方法将 args[0]字符串转换为数值，并以此生成 Float 对象。然后利用 Float 对象中的 floatValue()方法取出 float 型数据后保存到 f 变量中。

1.5 多态性的实现

面向对象程序设计语言的特点包括模块化(即数据抽象)、数据隐藏、继承和多态等。Java 作为面向对象程序设计语言也具有该类语言的显著特点。

在模块化方面，Java 语言程序都是由一个一个的类、接口及其对象组成的。在每一个对象中，既定义了相应的数据结构，同时又定义了操作这些数据结构的方法，这样，每一个对象都是一个完整的功能模块，都是对事物的高度抽象。

在数据隐藏方面，Java 语言中所定义的成员变量和方法，都是可以指定访问修饰符的，这些访问修饰符为对象的成员加上了访问控制权限，使得外部对对象中成员的访问是有限制的：对于那些私有成员来讲，外部就无法进行访问，从而起到了数据隐藏的作用。

在继承方面，Java 语言允许通过继承已有的类来生成新的类，从而能够减轻程序设计的工作量，提高程序的可靠性。

下面主要介绍一下 Java 语言中多态性的实现过程。

假如有如下两个类的定义：

```
class Mammal
{
    ⋮
}
class Cat extends Mammal
{
    ⋮
}
```

在此定义的基础上，首先定义 Mammal 类的变量 md：

```
Mammal md;
```

然后就可以将 Mammal 对象赋给变量 md：

```
md = new Mammal();
```

实际上，由于 Mammal 类是 Cat 类的父类，因此，也可以将 Cat 类的对象赋给变量 md：

```
md = new Cat();
```

此时的 Cat 类型被称为动态类型。这样，就可以利用 md 来调用其方法了，比如：

```
md.eat();
```

下面讨论一下这里所调用的 eat()方法应该是哪个类(Mammal 类或者 Cat 类)中的方法。

在 Java 语言中，如果所调用的方法 eat()仅在动态类型 Cat 中存在的话，则会出现编译错误(错误信息是在 Mammal 类中没有此方法)；如果所调用的 eat()方法仅在 Mammal 类中存在，则直接调用 Mammal 类中的方法；如果所调用的 eat()方法既在 Mammal 类中存在又在 Cat 类中存在，则将调用 Cat 类中的方法。由于只有在执行过程中才能决定动态类型，因此，md.eat()是执行 Cat 类中的方法还是执行 Mammal 类中的方法是在执行过程中决定的，这就是所谓的多态性。

需要注意的是，这里定义的 md 必须是父类的变量，否则将会出现错误。例如：

```
Cat mc;
mc = new Mammal();              //编译出错，出错信息为类型不一致
```

此时，即使进行强制类型变换也不行。例如：

```
mc = (Cat)new Mammal();         //出错
```

1.6 委托处理与功能继承

前面已经介绍过，在 Java 语言中只允许单一继承方式，也就是只允许有一个父类。这样，要想利用多重继承的功能，就必须采用另外的手段，其中委托(delegation)处理方式是比

较有效和常用的方法。由于在后续章节的学习过程中经常会使用委托处理方式，因此，在本节通过一个简单的例子来介绍一下委托处理方式。

下面定义一个简单的堆栈类，用于对字符串进行管理。

```
class StringStack
{
    protected int stackLen;
    protected int stackCount;
    protected String stack[ ];
    //构造方法
    StringStack(int len)
    {
        stackLen = len;
        stackCount = 0;
        stack = new String[stackLen];
    }
    //数据进栈处理
    public void push(String data)
    {
        if(stackCount>= stackLen)
        {
            System.out.println("Stack overflow.");
            System.exit(1);
        }
        stack[stackCount++] = data;
    }
    //数据出栈处理
    public String pop()
    {
        if(stackCount>0)
            return stack[--stackCount];
        else
            return "No data";
    }
}
```

下面再定义一个类 KeyValue，用于对关键字及其所对应的值进行管理。

```
class KeyValue
{
    protected int keyLen;
    protected int keyCount;
    protected String Key[ ];
    protected String Value[ ];
    //构造方法
    KeyValue(int len)
    {
        keyLen = len;                    //最大属性个数
        keyCount = 0;                    //已有属性个数
        Key = new String [keyLen];       //为 Key 申请空间
        Value = new String [keyLen];     //为 Value 申请空间
```

```
    }
    //返回 keyCount
    public int getCount()
    {
        return keyCount;
    }
    //存入 key 和 value
    public void addKeyValue(String key,String value)
    {
        if(keyCount<keyLen)
        {
            Key[keyCount] = key;
            Value[keyCount] = value;
            keyCount++;
        }
    }
    //返回指定序号的 key
    public String getKey(int number)
    {
        return Key[number];
    }
    //返回指定 key 的 value 值
    public String KeyToValue(String key)
    {
        for(int i = 0;i<keyCount;i++)
        {
            if(Key[i].equals(key))
                return Value[i];
        }
        return "";
    }
}
```

在上面两个类定义的基础上，定义类 StringStackWithKV，在该类中希望具有 StringStack 和 KeyValue 两个类的功能。由于 Java 语言只允许单一继承，因此，只能有一个类采用继承方式来使用，而另一个类则采用委托方式来继承其功能。下面给出 StringStackWithKV 类的定义。

```
class StringStackWithKV extends StringStack
{
    protected KeyValue delegation;
    StringStackWithKV(int length)
    {
        super(length);  //调用 StringStack 类构造函数
        delegation = new KeyValue(length);
    }
    public void addKeyValue(String key,String value)
    {
        delegation.addKeyValue(key,value);
    }
```

```
    public String getKey(int number)
    {
        return delegation.getKey(number);
    }
    public String KeyToValue(String key)
    {
        return delegation.KeyToValue(key);
    }
    public static final void main(String args[ ])
    {
        StringStackWithKV sKey = new StringStackWithKV(10);
        sKey.push("AA");
        sKey.push("BB");
        System.out.println(sKey.pop());
        System.out.println(sKey.pop());
        sKey.addKeyValue("A","Attribute");
        System.out.println(sKey.KeyToValue("A"));
    }
}
```

说明：尽管本程序是由三个类组成的，但每个类的定义都比较简单，很容易理解。这里，仅对委托处理方式进行简单的说明。

由程序的定义可知，StringStackWithKV 类继承了 StringStack 类，并在 StringStackWithKV 类的构造方法中，定义了一个 KeyValue 类的对象，并将其保存在 delegation 变量中。同时，在 StringStackWithKV 类中定义了与 KeyValue 类中同样的方法，只是这些方法体的处理过程完全是利用 KeyValue 类中的同名方法来实现的。在利用 StringStackWithKV 类的对象来访问这些方法时，都是利用 delegation 变量中所保存的对象来调用 KeyValue 类中的同名方法。因此，这种处理方式被称为委托处理，而 KeyValue 类则被称为委托类。

在后续章节中经常需要使用委托处理方式来完成多功能继承的任务。

第2章 分布式对象与CORBA

分布式对象是一门非常实用的技术，它能够解决不在同一个进程中的两个对象之间的调用问题。更重要的是，在分布式对象系统的支持下，这一调用过程是完全透明的，系统为程序设计人员屏蔽了非常复杂的调用过程，对程序设计人员来讲，对分布式对象的调用就好像是调用本地对象一样的简单。目前，可以利用很多技术来实现分布式对象功能，比如EJB、RMI和CORBA等，而CORBA是分布式对象技术的典型代表。本章主要介绍与CORBA有关的分布式对象的基本概念，主要包括CORBA系统的基本组成和基于CORBA分布式对象系统的处理过程等。

2.1 CORBA与OMG

CORBA是Common Object Request Broker Architecture的简称，即“通用对象请求代理结构”，它是由国际OMG(Object Management Group，对象管理组织)提出来的分布式对象系统设计规范。这也就是说，CORBA并不是一个具体的用于管理分布式对象的系统，更不是分布式对象应用系统，它只是一个分布式对象系统设计规范。目前，已经提出很多基于CORBA规范的分布式对象系统，比如ObixWeb、VisiBroker和JavaIDL等。

提出CORBA规范的国际OMG组织是1989年由企业组织成立的。初期只有8家公司左右(包括3Com、Canon、HP等)，到现在已超过800家公司。

2.2 CORBA的发展历程

CORBA从第一版公开发表开始到目前为止已经有十几年的历史了，其主要的发展历程如表2.1所示。

表2.1 CORBA的发展历程

版 本	时 间	主要功能
CORBA 1.0	1991.10	对象模型，静态/动态调用，C语言映射等
CORBA 1.1	1992.2	错误修正
CORBA 1.2	1993.12	错误修正
CORBA 2.0	1996.8	互操作性，C++语言、smalltalk语言映射，接口仓库扩展等
CORBA 2.1	1997.8	COBOL语言、Ada语言映射等
CORBA 2.2	1998.2	POA、Java映射等
CORBA 3.0	1998.9	非同步调用与QoS等

CORBA 规范的第 1.0 版是在 1991 年 10 月发表的。作为最初的规范，它确定了 CORBA 的基本框架。在第一版的规范中，包括 CORBA 的基本对象模型、静态对象调用和动态对象调用的基本过程、ORB 接口的定义、接口仓库以及 C 语言映射等基本内容。在此基础上，分别于 1992 年和 1993 年提出了 CORBA 1.1 版和 CORBA 1.2 版，在这两个版本中，主要是对最初发布的版本进行了错误修正。

在 CORBA 的第一版规范中(包括 CORBA 1.1 和 CORBA 1.2)有两个主要问题，一是在其规范描述中，有个别地方不明确，特别是对服务器的重要组成部分 BOA(Basic Object Adapter，对象适配器)的描述中有许多不明确的地方，这就导致了不同厂家的 ORB 产品所提供的 API 是有差别的，使得在一个 ORB 上开发的应用程序无法直接在另一个 ORB 产品上运行，从而不能保证应用程序的可移植性；二是没有对通信协议作出明确的规定，从而使得各 ORB 产品都实现自己定义的通信协议，无法实现各产品之间的互操作性(Interoperability)。CORBA 第一版中所出现的问题在其以后的版本中逐渐得到了解决。

CORBA 规范的第 2.0 版是在 1996 年 8 月发表的。在 CORBA 2.0 版本中规定了通信协议 IIOP(Internet Inter-ORB Protocol)，从而解决了各 CORBA 产品之间的互操作性问题。同时，在 CORBA 2.0 版本中还追加了 C++语言映射和 Smalltalk 语言映射的功能，从而扩展了能够用于开发 CORBA 分布式对象系统的语言范围，为用户提供了可选择的空间。在此基础上，对原有的接口仓库的功能进行了扩展，追加了能够写入接口仓库的 API。在 1997 年发布的 CORBA 2.1 版本中，追加了 wchar 字符类型和 wstring 字符串类型，同时规定了使用不同代码集的应用程序之间实现互操作性的处理机制，并追加了 COBOL 语言和 Ada 语言映射功能。在 1998 年发布的 CORBA 2.2 版本中，增加了三个主要功能：一是作为服务器重要组成部分的 BOA(Basic Object Adapter，基本对象适配器)被 POA(Portable Object Adapter，可移植对象适配器)所替代，确定了 POA 规范，从而很好地解决了一直以来困扰业界的可移植性问题；二是追加了 Java 语言映射功能，使得将 Java 与 CORBA 结合起来进行分布式对象系统的开发变得容易起来，这也是近几年对 CORBA 的研究与应用更加深入和普及的原因；三是实现了 CORBA 与 COM(Component Object Model，组件对象模型)的相互操作功能。

CORBA 规范的第 3.0 版是在 1998 年 9 月发表的。在 CORBA 3.0 版本中，主要追加了如下三种功能：

1) 对分布式组件的支持模型

在 CORBA 服务器设计过程中，通常会遇到这样的问题，这就是在 CORBA 规范中，并没有给出分布式组件的设计模式与标准，也没有给出组件与服务器之间的接口标准，这就导致了在一个应用服务器上开发的组件难以在另一个服务器上使用的问题。另外，在此之前的 CORBA 规范中只提供了一些低级的 API，这样，应用程序开发中的大部分工作都花费在事务处理、安全性以及生命周期管理等系统级的程序设计方面，为了解决这样的问题，提出了 CORBA 组件模型。支持 CORBA 组件的开发工具能够自动生成系统级的代码，这样，程序设计人员可专注于业务逻辑的开发。

2) 与 Java 及 Internet 的统合

在 CORBA 2.2 版本中，对从 IDL(Interface Definition Language，接口定义语言)到 Java 语言的映射进行了规定，使得利用 Java 语言进行 CORBA 应用系统的开发变为可能。

在 CORBA 3.0 版本中，进一步追加了从 Java RMI 接口到 CORBA IDL 接口的映射规定，使得在 CORBA 客户端中可以调用利用 Java RMI 开发的纯 Java 分布式对象，同时在 Java RMI 客户端中也可以调用 CORBA 分布式对象。目前提供这一功能的产品已经出现。除此之外，在 CORBA 3.0 版本中还规定了防火墙的规范，这是由于在通过 Internet 向外界发布基于 CORBA 的应用系统时，通过利用防火墙来进行访问控制是非常必要的，在 CORBA 3.0 中对防火墙的规范进行了标准化。

3）非同步调用与 QoS

在 CORBA 3.0 版本中，提出了异步调用方式和 QoS(Quality of Service，服务质量)的概念。在 CORBA 3.0 以前的版本中，对 CORBA 对象的调用一般都是采用同步调用方式，也就是说，客户端在发出调用请求以后就处于阻塞状态，直到获得来自服务器的返回信息为止。与此相对应，异步调用方式的处理过程是，客户端在发出调用请求以后立即进行其后续程序的执行，并不等待服务器程序的运行结果。在从服务器返回应答信息后，将异步调用客户端的相关处理代码进行处理。实际上在 CORBA 3.0 以前的版本中也包括 oneway 调用方式和延迟同步调用方式等，对这些调用方式进行合理的利用也能够实现与异步调用方式相同的功能。但是，在 CORBA 3.0 中，对这种异步调用方式进行了扩展，提出了被称为 TII (Time Independent Invocation，时间无关性调用)的异步调用方式，这种调用方式首先将请求信息加入队列中，从而能够保证即使通信线路或服务器临时出故障，也有可能完成对请求信息和应答信息的处理，同时由于信息的传递与队列操作都是在事务的控制下进行的，因此，能够提高信息传递的可靠性。除了异步调用功能之外，在 CORBA 3.0 版本中还追加了 QoS 功能。在 CORBA 系统中，所谓 QoS 是指将对象调用的请求或应答信息在客户端和服务器之间进行传递时的通信服务的质量。在 CORBA 3.0 以前的版本中，也可以指定相应的 QoS。比如，在安全服务中，就可以指定在信息传递时需要对信息进行加密以保证其私密性以及通过对信息进行校验来保证在传递过程不被篡改等的 QoS 服务。而在 CORBA 3.0 版本中，在已有的 QoS 的基础上，可以更详细地指定 QoS。比如，可以以请求信息和应答信息为单位来指定优先顺序，可以指定信息传送时间和超时时间等。

2.3 分布式对象的定义与特点

从表面上来看，分布式对象好像是很复杂的一个概念，但实际上它是实现非常单纯的想法的技术，从应用的角度来讲也并不复杂，但对分布式对象的实现却是需要非常复杂的处理过程的。

本节将首先给出分布式对象的定义，在此基础上，详细介绍分布式对象的特点。

2.3.1 分布式对象的定义

在不进行分布式处理的对象系统中，所有的功能都是在一个进程中实现的。在这种情况下，对象的调用者和被调用者都是在同一个进程的地址空间中配置的。如果不考虑访问权限的话，在一个进程中，任何对象都可以通过引用来访问另一个对象。

图 2.1 给出了同一个进程中的对象调用过程。这里，对象 1 和对象 2 以及作为调用者

的主程序都位于同一个进程(或程序)中。

由于在图 2.1 的对象系统中,调用者和被调用者都位于同一个进程中,因此调用过程非常简单,这种处理方式称为本地对象系统。

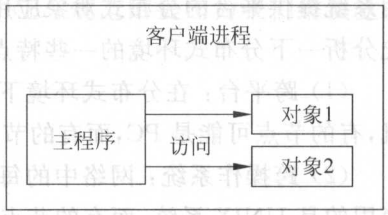

图 2.1　本地对象系统

与上述处理过程相对应,如果要调用的对象和调用者不在同一个进程中,则称为分布式对象系统,其中作为被调用者的对象被称为分布式对象或远程对象。

图 2.2 给出了分布式对象系统的示意图。这里,由于客户端进程要调用的对象位于服务器进程中,因此,就成为分布式对象系统了,其中的对象 1 和对象 2 被称为分布式对象。

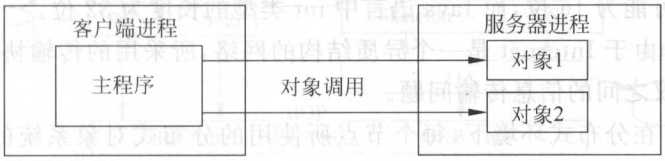

图 2.2　分布式对象系统

对于分布式对象系统来讲,既可以是同一个 OS 上的多个进程之间的分布式,也可以是多个机器上的(多个 OS)多个进程之间的分布式,尽管其实现难度可能有很大差异,但基本结构是相同的。

2.3.2　分布式对象系统的透明性

CORBA 采用了分布式计算模型,而分布式计算模型具有以下特点。

(1) 分布性:所谓分布性是指数据、处理和控制等并不驻留在某个单一站点上,而是均匀地分布在整个网络的每个节点上。

(2) 并行性:在分布式计算模型中,绝大部分计算都可以并行进行。

(3) 透明性:从用户应用的角度来讲,分布式计算模型中的复杂的处理过程都是透明的。对分布式对象的调用来讲,用户就如同调用本地对象一样来调用分布式对象。

(4) 共享性:在分布式计算模型中,网络中的软、硬件资源高度共享。

(5) 鲁棒性:由于分布式计算模型中的数据、处理和控制等都被均匀地分布在网络中的各个节点上,因此,单个节点的故障并不能引起整个系统的瘫痪。

在分布式计算模型的上述特点中,对用户来讲"透明性"是其需要考虑的重要因素之一。分布式对象系统的重要之处就在于"尽量使开发者没有意识到分布式这一事实来实现分布式系统"。这也就是说,对于客户端应用来讲,就像使用本地对象一样来使用分布式对象。显然,透明性越高越好,但是分布式处理的透明性越高,实现起来就越复杂,这一点将在下一节中详细讨论。

2.3.3　分布式对象系统的复杂性

1. 主要特点

这里所说的分布式对象系统不是指分布式对象应用系统,而是指为开发分布式对象应

用系统提供平台的分布式对象应用系统开发环境。为了说明分布式对象系统的复杂性，首先分析一下分布式环境的一些特点。概括来讲，分布式环境具有如下一些主要特点。

(1) 跨平台：在分布式环境下，每个节点的构成可能都是不同的，有的节点可能是大型机，有的节点可能是PC，而有的节点可能只是一台笔记本计算机等。

(2) 跨操作系统：网络中的每个节点所使用的操作系统可能是不同的，有的节点可能使用的是UNIX系统，而有的节点可能使用的是Windows系统等。

(3) 跨语言：每个节点所使用的计算机语言可能不同，对于分布式对象来讲，完全可以采用不同的语言来实现客户端和服务器。比如，客户端利用Java语言实现，而服务器则采用C++语言实现，由于语言不同，在某些处理方面可能会产生差异，比如，C++语言中的int类型变量的长度可能为16位，而Java语言中int类型的长度为32位。

(4) 跨协议：由于Internet是一个异质结构的网络，所采用的传输协议可能是不同的，这将导致不同协议之间的信息传输问题。

(5) 跨版本：在分布式环境下，每个节点所使用的分布式对象系统的版本可能是不同的，这就导致了不同版本之间的兼容性问题。

分布式环境的上述特点决定了分布式对象系统的复杂性。如果说得更明确一点的话，下述两个因素决定了分布式对象系统的复杂性。

2. 复杂性

(1) 分布式环境的性质决定了分布式对象系统的复杂性。

在这一点上，最容易理解的是对象的定位(寻址)问题。由于本地对象系统都是在同一个地址空间中寻址的，因此处理起来比较方便，而分布式对象系统不是在同一个进程的地址空间中寻址的，这样，要想调用分布式对象的话，就必须要了解分布式对象所在的服务器进程信息(包括对象所在的网络上的主机地址和确定特定进程的端口号)以及服务器进程上的分布式对象的识别信息等。同时，对于垃圾收集和安全性等方面都是必须要考虑的重要因素。

(2) 节点环境的多样性决定了分布式对象系统的复杂性。

这里的“环境”是指每个节点所使用的操作系统和程序设计语言等。在分布式环境下，每个节点可能使用不同的操作系统，比如有的节点可能使用UNIX和Linux操作系统，而有的节点则可能使用Windows系统。同时应用软件的开发也可以利用多种语言，特别是对于分布式对象系统来讲，客户端可能采用Java语言开发，而服务器则可能采用C++语言开发，在这种情况下将会出现一些类型匹配等问题。比如，就int类型来讲，在Java语言中int类型的长度是32位，而在C++语言中，所使用的平台不同其int类型的长度也可能不同，这样，在客户端调用服务器的分布式对象时就不能正确地进行数据的传输和存取，因此，就需要采用某些通用的规则来约束才行。

另外，在分布式对象的调用过程中所进行的参数传递和异常处理等也都是必须要认真考虑和解决的问题。尽管通过对所使用的环境和语言进行限制可以解决其中的一些问题，比如，Java中的分布式对象系统RMI，由于只支持Java一种语言，因此，类型匹配问题也就得到了解决，但它在解决了一些问题的同时，也限制了某些应用和网络的多样性特点。因此，要开发一个适合于分布式环境下使用的分布式对象系统开发平台是很困难和复杂的事

情，同时也是一件非常有应用价值的事情。在这方面，CORBA 规范的提出为我们解决这类问题开辟了新的领域。

2.4　CORBA 系统的基本构成

基于 CORBA 的分布式对象系统的基本构成如图 2.3 所示。图中的左边是 CORBA 客户端的基本组成，右边是服务器的基本组成。

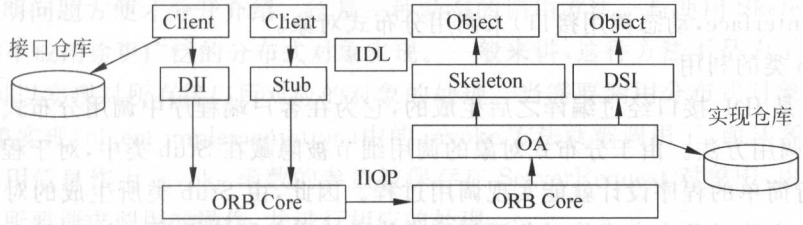

图 2.3　CORBA 的基本构成

下面对 CORBA 系统的基本组成部分及其相关概念进行解释。

1. IDL

IDL(Interface Definition Language，接口定义语言)是用于描述分布式对象接口的定义语言，通过 IDL 定义之后，就使得客户端和服务器之间的接口明确下来了，这样，有利于独立开发客户端和服务器程序。一般来讲，在分布式对象系统中，作为客户端的应用程序和作为服务器的分布式对象都是独立开发和执行的，这样，为了使两者的接口一致，CORBA 提供了接口定义语言 IDL。分布式对象系统设计的第一步就是利用 IDL 来定义客户端和服务器都需要的接口，换句话说，能够被客户端所调用的分布式对象中的方法一定是在由 IDL 所定义的接口中所声明的方法。

在程序中是不能直接使用 IDL 接口信息的，而是需要对 IDL 接口进行编译后使用。IDL 接口经过编译之后，就被映射为相应的语言了。现在可被映射的语言包括 Java、C++、C、Smalltalk、Ada 和 COBOL 等。如果说可以利用某种语言来开发基于 CORBA 的分布式对象系统的话，那一定存在可将 IDL 接口信息编译为那种语言的编译器。IDL 接口经过编译之后，将生成 Stub 和 Skeleton 等类，这些类将被用于开发客户端的应用程序和服务器的分布式对象。因此，为了利用 IDL 接口信息来进行程序设计，就必须要了解 IDL 接口定义是如何被映射为开发应用程序所使用的程序设计语言的。

下面这段代码就是一个简单的 IDL 接口定义，其中声明了一个接口 xxx、一个只读属性 yyy 和两个操作(或称方法、函数等)aaa 和 bbb。这里，zzz 是模块的名字。

```
module zzz
{
    interface xxx
    {
        readonly attribute long yyy;
        long aaa(in long a);
```

```
        long bbb(in long b);
    };
};
```

有关 IDL 的基本结构和定义形式，将在后续章节中详细介绍。

2. 静态 Stub 和 DII

在利用 CORBA 系统来实现分布式对象系统时，对客户端应用的实现可以采用两种方式，一种是利用 Stub（一般被称为“桩”）类来调用分布式对象，另一种是利用 DII（Dynamic Invocation Interface，动态调用接口）来调用分布式对象。

1）Stub 类的利用

Stub 类是 IDL 接口经过编译之后生成的，它为在客户端程序中调用分布式对象提供了一种静态的调用方法。由于分布式对象的调用细节被隐藏在 Stub 类中，对于程序设计者来讲，只要进行简单的程序设计就能实现调用过程。因此，由 Stub 类所生成的对象一般被称为代理对象，由该对象来完成将对象调用请求传递给服务器等操作。由于 Stub 类是由 IDL 接口经过编译之后产生的，因此在程序设计时需要获得分布式对象的接口定义。实际上，对于客户端程序来讲，绝大部分应用都可以通过利用 Stub 类来实现。

2）DII 功能的利用

尽管在图 2.3 中是将 DII 与 ORB Core 分开的，但实际上 DII 也是 ORB 功能的一部分，只是为了说明问题方便才分开介绍。这是一种动态的调用方法。在应用程序中来组成调用分布式对象的请求。这种方法是在程序运行过程中从接口仓库中来获取分布式对象的接口定义信息，并据此来生成调用请求。这样，由于 IDL 接口经过编译之后其信息已经被保存在接口仓库中，因此不需要事先获取接口定义。使用 DII 在某种程度上能够设计出通用的客户端程序。

3. 接口仓库

接口仓库（interface repository）用于保存分布式对象的接口定义信息。由 IDL 定义的接口信息，经过编译后被保存在 Stub 和 Skeleton 中，因此，使用 Stub 和 Skeleton 来编程时，就不一定需要接口仓库。接口仓库主要用于使用 DII 和 DSI 编程的情况。

在接口仓库中对接口定义信息的保存方法是与实际的 ORB 产品有关的。然而，从应用程序的角度来看，组成接口定义信息的模快（module）、接口（interface）、操作（operation）、属性（attribute）、常量（const）、异常（exception）以及用户定义的数据类型（比如 struct）等各种定义信息，都可以作为 CORBA 对象进行访问。接口仓库中的信息一般是在编译 IDL 接口定义时被保存的。

4. 静态 Skeleton 和 DSI

在利用 CORBA 系统来实现分布式对象系统时，OA（对象适配器）可以使用两种方法将来自客户端的对象调用请求传递给分布式对象实现：一种是利用由 IDL 接口定义编译生成的 Skeleton（一般被称为“骨架”）的静态方法，另一种是利用 DSI（Dynamic Skeleton Interface，动态骨架接口）的动态方法。

1) Skeleton 类的利用

Skeleton 类是 IDL 接口定义经过编译之后所生成的，服务器程序可以通过继承 Skeleton 类来实现分布式对象的功能。在这种情况下，由 OA 来判断客户端要调用哪个接口中的哪个对象中的哪个方法，并将调用请求传递给该方法。对于分布式对象的开发者来讲，就可以不必考虑分布式方法的具体调用过程，而可以把主要精力都集中在分布式方法的具体实现上。

2) DSI 功能的利用

尽管在图 2.3 中是将 DSI 与 ORB Core 分开，但实际上 DSI 也是 ORB 功能的一部分，只是为了说明问题方便才分开介绍。这是一种动态的调用方法。与使用 Skeleton 相比，使用 DSI 能够生成用途更广泛的分布式对象实现。一般来讲，这种方法不是为了实现特定的接口，而是可以实现对所有接口所对应的对象的处理。当需要调用分布式对象中的方法和属性时，对象实现(object implementation)中的 invoke 方法就被调用了，而从客户端传递过来的对象调用信息作为 invoke 函数的参数被保存在 ServerRequest 对象中，由对象实现类本身来判断所要请求调用的操作，并进行相应的处理。

5. 实现仓库

实现仓库(implementation repository)用于保存启动服务器进程所需要的一些必要信息。服务器端的 OA 在接收到来自客户端的对象调用请求之后，如果所要调用的对象的服务器进程还未启动的话，则需要根据相应的设置来自动启动该对象所在进程。在这种情况下，应用程序的开发者必须预先将服务器名以及启动服务器所必需的一些信息(如可执行文件的路径、Java 的类名以及命令行参数等)以服务器为单位进行登录，用于保存这些信息的就是实现仓库。

与接口仓库不同，在 CORBA 规范中并没有给出保存在实现仓库中的信息的具体内容以及用于访问实现仓库的 API 等定义，因此实现仓库的具体实现是与实际的 CORBA 系统有关的。

6. 分布式对象实现

在服务器上实现的分布式对象功能的本地对象，被称为分布式对象实现。由分布式对象实现来完成来自客户端的对象调用请求。一般来讲，一个分布式对象实现对应一个 IDL 定义中的一个 interface 定义。

概括地讲，分布式对象实现主要是由以下几部分组成的。

(1) 分布式方法：与 IDL 定义中的接口相对应的分布式方法的实现部分。它是分布式对象实现的核心部分，这部分方法的定义与 IDL 定义中的某个接口中定义的操作是一一对应的。IDL 接口中所声明的分布式方法是在该部分中给出具体实现的，也只有在这里定义的方法才能作为分布式方法被客户端所调用。

(2) 本地方法：本地方法不能由客户端来调用，它一般是由分布式方法来调用的，以辅助分布式方法来实现相应的功能。本地方法在 IDL 定义中没有对应的定义。

(3) Skeleton 类：Skeleton 类的作用是接收来自客户端的调用请求并启动分布式方法。它是由 IDL 接口定义经编译后自动生成的。分布式对象实现通过继承 Skeleton 类，将其组

合到分布式对象实现类中。

7. 分布式对象引用

为了利用分布式对象，就必须首先获取分布式对象引用。显然，分布式对象的引用与常用的本地对象的引用是不同的，但就其作用来讲，分布式对象的引用如同本地对象的引用一样，也是为了指定特定的分布式对象。

以 Java 语言为例，本地 Java 对象的引用就是内存的地址，由于调用者和被调用者都存在于同一个进程之中，因此，只要获取了本地对象所在的内存地址，也就可以唯一地确定该对象。对于分布式对象来讲，其引用的确定过程就没有这么简单了。由于调用者和被调用者不在同一个进程中，因此单纯靠内存地址是无法唯一确定分布式对象的。下面，考察一下分布式对象引用中应该包含哪些主要信息。

了解 socket 编程知识的读者都知道，要识别一个进程，既需要知道进程所在的服务器地址，又需要知道进程所在的端口号，这样才能唯一地确定一个进程。与识别一般进程不同的是，对于分布式对象来讲，只了解其所在的进程还不行，还必须要了解是该进程中的哪个分布式对象，同时还需要了解该分布式对象所对应的 IDL 定义信息，以便能够根据所获得的分布式对象引用来生成客户端程序所要利用的代理对象，进而通过代理对象来实现请求信息的传递等功能。

基于上述原因，分布式对象引用中应该包含如下一些基本信息：

(1) 分布式对象实现的定位信息；

(2) 分布式对象接口的数据类型；

(3) 分布式对象的其他附加服务的信息。

由上述介绍可知，分布式对象的定位信息是由两部分组成的：一是服务器进程的定位信息(地址)，二是为了指定服务器进程上特定的分布式对象实现而设置的对象键(object key)。

服务器进程的定位信息，也就是服务器进程的地址是与所使用的网络系统有关的，对于 TCP/IP 网络来讲，服务器进程的地址是由唯一确定一台机器的 IP 地址和唯一确定该机器上的某个进程的端口号(port 号)组成的。对象键是用于识别一个进程上所存在的分布式对象而设置的任意数据。对象键既可以在服务器进程中指定，也可以由 ORB 自动生成。根据服务器进程的地址，就可以通过网络系统在特定的服务器进程之间建立连接。同时，根据对象键，就可以指定服务器进程上的特定的分布式对象实现。因此，根据这两个信息，客户端应用就可以将启动方法的请求传递给分布式对象实现中。

所谓“分布式对象接口的数据类型”，是指由 IDL 定义的分布式对象接口的数据类型。该数据类型信息将用于决定利用哪个 Stub 类在客户端中生成代理对象。前面已经介绍过，Stub 类作为代理对象，将完成将请求信息传递给服务器等功能，从而将为程序设计人员屏蔽掉一些烦琐而复杂的工作。客户端在获得分布式对象引用之后，将利用该引用来生成 Stub 类的代理对象，“分布式对象接口的数据类型”将为生成哪个 Stub 类的代理对象提供识别信息。

“附加服务的信息”是指在利用分布式对象时，作为前提条件的一些附加服务的信息。比如，分布式对象的安全性以及事务处理等相关信息等。这些附加服务的信息是在调用分布式对象的方法时被附加提供的，因此，也将作为分布式对象信息的一部分被保存在分布式

对象引用中。

8. ORB

ORB(Object Request Broker,对象请求代理)是基于 CORBA 的分布式对象系统的核心,在代理对象(Stub)和分布式对象实现的背后,由 ORB 提供了实现分布式对象系统所需要的各种服务。概括地讲,由 ORB 提供的服务包括如下四种:

1) ORB 接口

由客户端和服务器共同利用的 ORB 的服务都在 ORB 接口中进行了声明。主要包括 ORB 的初始化(获取 ORB 对象的引用)、获取 CORBA 服务对象、动态调用请求的生成以及分布式对象引用的变换(将分布式对象引用变换为字符串或相反的变换)等。

2) DII

如前所述,DII 是动态启动接口,在将客户端生成的对象调用请求传送给 ORB 时需要利用 DII。在客户端应用程序中可以直接利用 DII 的功能,同时在 Stub 类中也是利用 DII 来进行分布式方法调用的。

3) DSI

如前所述,DSI 是动态骨架接口,是与 DII 对应的服务器端的接口。分布式对象实现利用该接口从 ORB 中获取来自客户端的请求。在分布式对象实现中可以直接利用 DSI 的功能,同时在 Skeleton 类中也是利用 DSI 来获取分布式方法的调用请求的。

4) OA

OA 是对象适配器,它是用于在服务器进程中管理分布式对象而提供的接口。OA 提供的主要功能包括分布式对象引用的生成以及将分布式对象实现登录到 ORB 中等。在 CORBA 2.2 以前使用的是 BOA(Basic Object Adapter,基本对象适配器),由于这种 BOA 的可移植性不好,所以在 CORBA 2.2 以后的版本中由 POA(Portable Object Adapter,可移植对象适配器)取代了 BOA,以提高系统的可移植性。

9. IIOP

CORBA 采用的上层协议为 GIOP(General Internet Inter-ORB Protocol),其中包括信息交换规则、信息格式定义以及数据的表示规则等。CORBA 的下层协议为 IIOP(Internet Inter-ORB Protocol),由 IIOP 协议将 GIOP 协议映射为 TCP/IP 协议。CORBA 系统的协议为实现各种不同 ORB 产品的互操作性提供了强有力的支持。

2.5　CORBA 分布式对象环境

CORBA 分布式对象环境是由 CORBA 和 CORBAServices 等构成的,如图 2.4 所示。

由图 2.4 可知,CORBA 分布式环境中的各组成部分是由 ORB 联系起来的,以 ORB 为核心,包括 CORBAServices、CORBAfacilities、产业领域接口和应用程序对象等几部分。

应用程序对象(application object)是利用 CORBA 开发的分布式对象集合;产业领域接口(domain interface)是每个产业领域的通用接口的集合,是已经为某些领域开发完成的分布式对象的通用接口;CORBAfacilities(CORBA 通用工具)是一组为应用程序开发提供

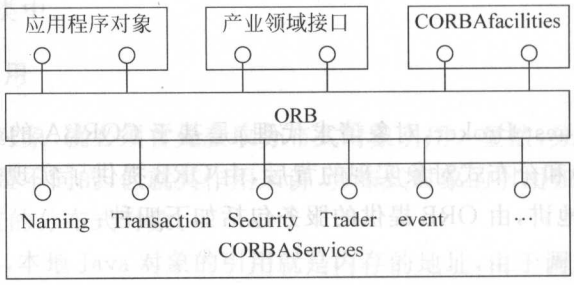

图 2.4 CORBA 分布式对象环境

的通用服务，可以包括与用户接口有关的工具、与信息管理有关的工具以及与系统管理有关的工具等。

CORBAServices(CORBA 通用对象服务)为开发分布式对象系统提供了很多基本服务。这些基本服务为 CORBA 分布式对象系统的开发提供了有力的支持。所有的服务对象都是可以通过 ORB 对象中的方法来获得的。CORBA 的基本服务主要有如下几个。

1. 命名服务

对象命名服务(naming service)是指为登录和检索分布式对象引用而提出的服务。与对象洽谈服务不同，它是以名字来进行登录和检索的。

2. 事件服务

事件(event)服务支持基本的事件通信机制。一个对象可以通过两种方式来处理事件：

(1) 一旦事件发生，主动向有关对象发送消息。在 CORBA 中，这种方式被称为 push(推)方式。

(2) 事件发生后，在被有关对象询问时再发送消息。在 CORBA 中，这种方式被称为 pull(拉)方式。

3. 持续对象服务

一般来讲，对象在运行过程中有两种状态：

(1) 动态状态：对象的所有信息主要保存在内存中，一旦程序结束或关机，对象的状态信息就会消失。

(2) 持续状态：对象的所有信息主要保存在硬盘等存储介质中，对象的状态能够长久保持。

持续对象服务(persistent object service)就是提供将持续对象的数据保存在数据库中的机制。

4. 对象生存期服务

对象生存期服务(life cycle)用于管理对象的生存周期。主要包括对象的创建、释放、复制、移动和删除等操作。

5. 对象关系服务

对象关系服务(relationship)主要用于管理对象之间的关系。比如,对象之间是“拥有”关系还是“隶属”关系等。

6. 对象外化服务和对象内化服务

对象外化服务(externalization)和对象内化服务(internalization)主要用于将对象的状态写入“流”中(外化)或从“流”中读入对象的状态(内化)。

7. 事务服务

事务服务(transaction)主要提供分布式事务服务机制。

8. 并发控制服务

并发控制服务(concurrency control service)提供对象的排他控制服务。也就是为分布式对象或分布式事务提供一种合理的机制,以便使它们能够成功读取和修改共享资源,而不发生读错数据或丢失修改等错误。

9. 对象安全服务

对象安全(security)服务提供分布式安全服务功能。在 CORBA 中,将安全服务移入 ORB 中,使问题变得容易解决。

10. 对象时间服务

对象时间(time)服务提供时间同步和时间查询的功能。

11. 对象许可服务

对象许可服务提供软件的使用证书(license)管理功能。它允许对象开发者为对象发放使用证书,只有获得许可的用户才可使用对象提供的服务。

12. 对象查询服务

对象查询(query)服务提供对象检索的功能。可以检索任何 CORBA 对象。

13. 对象属性服务

对象属性(property)可以在 IDL 接口定义时进行定义,但这种定义一经编译之后就不能修改了。对象属性服务提供了在执行时对对象的属性进行设定的功能。

14. 对象包容服务

对象包容服务(collection service)为对象提供各种各样的容器,包括队列、堆栈、数组和集合等。

15. 对象洽谈服务

对象洽谈服务(trader service)提供以服务内容来登录对象和检索对象的服务。

上面介绍的是CORBA能够提供的一些基本服务。实际上,CORBA系统不同,所能提供的服务种类也是有差异的。

2.6 分布式对象系统的处理过程

上面介绍了基于CORBA分布式对象系统的构成元素。在本节中将说明这些元素是怎样协作来实现分布式对象方法的调用过程的。

图2.5给出了从“分布式对象实现的生成与登录”开始到“分布式方法调用并返回调用结果”为止的一系列处理过程。

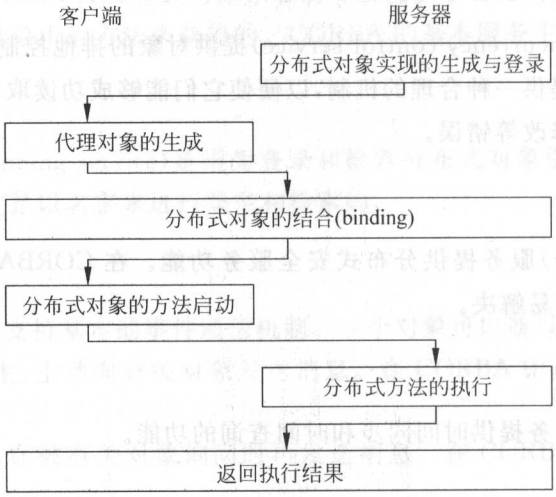

图2.5 分布式对象系统的处理过程

由图2.5可知,分布式对象系统的处理过程是由分布式对象的生成与登录、代理对象的生成、分布式对象的结合、分布式方法的启动、分布式方法的执行以及执行结果返回等几部分组成的。下面详细介绍分布式对象系统的每一个处理过程。

1. 分布式对象实现的生成与登录

由图2.5可知,分布式对象实现的生成与登录是整个处理过程最先执行的,一系列的处理过程是从分布式对象实现的生成与登录开始的。这一处理过程主要是由ORB的初始化、分布式对象实现的生成、分布式对象引用的生成、将分布式对象实现登录到ORB中、接收来自客户端的调用请求以及将分布式对象引用输出到服务器进程的外部等几部分组成的。

1) 分布式对象实现的生成

在服务器进程中,首先为了使用ORB而进行初始化操作之后,就需要进行分布式对象实现的生成操作。由于在服务器进程中,分布式对象实现相当于本地对象,因此,服务器端

的分布式对象生成如同本地对象生成一样，可以通过调用分布式对象实现类中的构造函数来生成分布式对象实现(也就是分布式对象)。

分布式对象生成之后，就需要生成分布式对象引用。在ORB的接口中提供了生成分布式对象引用的功能，程序中可以直接利用。实际上，由于多数CORBA产品在生成分布式对象的同时也自动生成了分布式对象引用，这样，应用程序的开发者不需要显式地生成分布式对象引用。

在前面已经介绍过，分布式对象引用中包含三种信息(分布式对象实现的地址、分布式对象接口的数据类型和分布式对象的附加服务)，这些信息的设置是由ORB的功能来实现的。

2) 将分布式对象实现登录到ORB

分布式对象实现生成之后，就需要将其登录到ORB中。

将分布式对象实现登录到ORB的操作是通过利用OA提供的接口、由服务器应用来进行的。将分布式对象实现登到ORB的目的是为了由ORB来管理分布式对象实现的信息，以便在以后从客户端应用中获取启动方法的请求时，可以利用其管理的信息来决定应该调用的分布式对象实现。

在ORB中管理分布式对象的方式依赖于相应的CORBA产品，也就是说CORBA系统不同其管理方式也可能不同。但不管如何管理，只要在ORB中建立了如图2.6所示的对象键(object key)与分布式对象实现的本地引用的对照表即可。有了这样的对照表，ORB在接收到来自客户端的调用请求(其中包含对象键信息)之后，就可以检索到与其对应的分布式对象了。

对象键	本地对象引用
对象键	本地对象引用
对象键	本地对象引用

图2.6　对象键与本地对象引用的对照表

上述过程完成之后，对分布式对象的方法启动的准备工作就结束了，此时，就可以开始接收来自客户端应用的启动方法的请求。为了将准备工作结束这件事通知ORB，服务器应用通过利用OA提供的接口来完成。这时，服务器应用就进入等待来自客户端的对象方法启动的请求的循环之中。由于客户端能够调用分布式对象的前提是需要获取分布式对象引用，因此，在服务器的准备工作结束之前，还需要将所生成的分布式对象引用输出到客户端能够访问到的地方(比如文件等)。

3) 分布式对象引用的导出

为了使客户端应用能够获得分布式对象引用，服务器端需要对分布式对象引用进行导出(export)操作。分布式对象引用的导出操作包括命名服务操作和利用文件等各种方法，但其基本处理过程也就是将分布式对象引用Stream化(流式化)，并存放到服务器进程的外部。

在上述工作的基础上，服务器端的准备工作就全部结束了，这时，根据需要客户端就可以提出调用分布式对象的请求了。为了简化程序设计过程，该请求的发送可通过代理对象来完成。

2. 代理对象的生成

在服务器端准备好的前提下，为了简化处理过程，提高程序设计效率，在分布式对象系统中，客户端要做的第一件工作就是创建代理对象，由代理对象来实现分布式对象的调用过程。该处理过程是由ORB的初始化、分布式对象引用的导入以及对所获取的分布式对象

进行 narrowing 处理等几部分组成的。

1）分布式对象引用的导入

对于客户端应用来讲，其最初的工作也是进行 ORB 初始化。然后，就是获得分布式对象引用。这一处理过程同导出(export)相对应，被称为分布式对象引用的导入(import)。

分布式对象引用的导入过程与导出的方式有关，但其基本的处理过程就是读入已被 stream 化的信息并再现出原来的分布式对象引用。

2）narrowing 处理

所谓 narrowing 处理，就是缩小对象的引用范围。即使分布式对象引用能够在客户端进程中再现出来，也不能直接利用该引用来启动分布式对象中的方法，这是由于对象的调用者和对象本身并不在同一个进程中的缘故。为了启动分布式对象中的方法，必须根据所获取的分布式对象引用来生成 Stub(桩)类的代理对象。这一处理过程被称为 narrowing。

一般来讲，在 CORBA 系统中，narrowing 处理主要用于这样两个操作：一是根据所获取的分布式对象引用来生成 Stub 类的代理对象，使得以后对分布式对象的调用过程都是通过代理对象来完成的；二是当在程序中需要利用 CORBA 提供的一些服务时，就需要首先获取这些服务对象，而所获取的这些服务对象的类型都是 org. omg. CORBA. Object，即所有对象的父对象。为了利用特定的服务对象，就需要利用 narrowing 操作将其转换为特定的服务对象。

通常，在 CORBA 系统中，narrowing 操作一般是在相应接口经过编译映射之后所生成的 Helper 类中提供的。

3. 分布式对象的结合(binding)

在客户端获取了分布式对象引用并正确地生成了代理对象以后，就需要在客户端的代理对象和服务器的分布式对象实现之间建立连接(connect)。通过这一连接，将启动方法的请求从客户端传递给分布式对象实现。在 CORBA 中，把建立连接的处理称为 binding。binding 处理是在生成了代理对象以后，在最初的方法启动之前的某个时候进行的。实际上应在何时进行连接，依赖于 CORBA 产品的具体实现。

分布式对象的结合过程主要是由客户端的端口生成、物理连接的建立以及分布式对象实现存在与否的检查等几部分组成的。

分布式对象的结合过程主要由以下几部分组成。

(1) 代理对象在客户端应用内部生成端口 Port。

(2) 代理对象根据存放在分布式对象引用中的服务器进程地址信息，从客户端的端口到服务器进程的端口建立网络上的物理连接。

(3) 建立了物理连接之后，利用此连接来检查分布式对象实现是否存在。为了进行这一检查，代理对象将存放在分布式对象引用中的对象键(object key)发送给服务器进程。

服务器进程将代理对象传递过来的对象键与其管理的对象键和分布式对象实现的本地引用的对照表进行比较，以便检查与对象键对应的分布式对象实现是否存在。

经过以上的处理，从代理对象向分布式对象实现发送的启动方法的请求就可以被确认下来，同时在两者之间建立了连接。

4. 分布式对象的方法启动

客户端与服务器之间的 binding 处理完成以后，就可以启动分布式对象的方法了。客户端进行方法启动的处理过程主要是由代理对象的方法启动、请求对象的生成、相关信息设置、将请求对象进行 marshaling 操作生成请求信息以及将请求信息发送给服务器等几部分组成的。在此过程中，出现了几个重要的概念，包括请求对象和 marshaling 操作等。

(1) 客户端应用如同调用本地对象一样来调用代理对象中的方法，即开始执行代理对象中的相应方法。这时，在代理对象的方法内部来生成请求对象(request object)。

(2) 将来自客户端应用的参数等信息从本地(local)变量中取出来，设置到请求对象中。必须设置的信息包括：

- 分布式对象引用的对象键；
- 方法名；
- 所需要的参数值。

(3) marshaling 操作。marshaling 是排列整齐的意思。在分布式环境下，各个节点的特性是有差异的，这既体现在操作系统方面，也体现在程序设计语言方面。这种差异最终都体现在数据类型的表现方面。因此，在为请求对象设置信息时，必须要考虑这种差异，也就是客户端平台与服务器端平台在数据表现形式上的差异。必须要考虑到如果操作系统和程序设计语言不同的话，数据类型的表现形式也可能不同这种情况，在前面我们提到的 C++语言与 Java 语言中的 int 类型的差异就是一个典型的例子。

因此，在 CORBA 中，为请求对象设置的信息在发送给服务器进程之前都要转换为对所有平台都通用的数据类型的表现形式，这一处理过程被称为 marshaling。此时所利用的通用数据的表现形式是由 CORBA 规范规定的。

(4) marshaling 操作结束之后，由请求对象中设置的信息来生成请求信息(request message)，并通过在代理对象和分布式对象实现之间建立的连接将其从客户端发送给服务器。此时，代理对象中的方法就处于等待从服务器进程返回执行结果的状态。

5. 分布式方法的执行

从客户端应用发送过来的请求信息，经过网络上的物理连接到达服务器进程之后，服务器端的方法调用过程就开始了。

服务器端的方法执行过程是由请求信息获取、将请求信息进行 unmarshaling 操作、确定所要调用的分布式对象、调用 dispatch 方法和调用分布式方法等几部分组成的。在此过程中，出现了几个重要的概念，包括 unmarshaling 和 dispatch 方法等。

1) unmarshaling 操作

unmarshaling 操作是 marshaling 操作的反过程。到达服务器进程的请求信息首先被传递给 ORB。由于该请求信息已经被转换为所有平台都通用的形式，这样在服务器的特定平台上并不能直接使用，因此，由 ORB 首先将请求信息转换为与服务器进程环境相对应的表现形式，这一处理过程与 marshaling 处理相反，故称为 unmarshaling。在此基础上，为了将请求信息传递给分布式对象实现，以 unmarshaling 之后的信息为基础生成服务器用的请求对象(request object)。

2）dispatch 方法的调用

ORB 根据存放在请求对象中的对象键来确定与其对应的分布式对象实现，分布式对象实现确定了以后，就启动 Skeleton 类中的 dispatch 方法。

Skeleton 类中的 dispatch 方法是对在 DSI 中定义的方法的实现，ORB 根据 DSI 的接口来启动 dispatch 方法。这时，ORB 将请求对象作为参数传递给 dispatch 方法。在 dispatch 方法中，从此请求对象中取出应该启动的分布式方法名和应该传递的参数值，并将其设置到本地(local)变量中。

3）分布式方法的启动

进行了上述处理之后，在 dispatch 方法中就启动(调用)分布式方法。由于是根据从请求对象中取出的分布式方法名来启动相应的分布式对象的，因此称为 dispatch 方法。

与此同时，把前面存放在本地变量中的参数传递给分布式方法(即作为分布式方法的参数)。由于对分布式方法的启动是同一个分布式对象实现内部的方法启动，因此，对分布式方法的调用执行过程与本地方法的调用执行过程是相同的。

6. 执行结果的返回

服务器端的分布式方法执行结束以后，就需要从服务器进程向客户端应用返回执行结果。从服务器向客户端返回执行结果是由服务器和客户端的两部分处理过程组成的。在服务器端所执行的一系列处理包括将执行结果保存到请求对象中、marshaling 操作后生成回答信息以及将回答信息返回给客户端等；在客户端所执行的一系列操作包括接收回答信息、unmarshaling 操作后生成请求对象以及将请求对象中的信息保存到本地变量中等。

(1) 分布式方法执行完了以后，首先将控制返回到 Skeleton 类中的 dispatch 方法中。在 dispatch 方法中，将应该返回给客户端应用的参数和函数返回值从本地变量中取出来，存入服务器端的请求对象中。

(2) 在 dispatch 方法中对请求对象的操作执行结束以后，将控制返回到 ORB 中。ORB 将对请求对象中的信息进行 marshaling 处理，然后，ORB 根据请求对象中的信息来生成回答信息(reply message)，并通过物理连接将其返回给客户端应用。

(3) 客户端应用的 ORB 接收来自服务器的回答信息并对其进行 unmarshaling 操作之后，将其存入客户端的请求对象中。然后，将控制返回到发送“启动方法请求”的代理对象的方法中。代理对象的方法从请求对象中取出参数值和返回值，并将其存入本地变量后，将控制返回到客户端应用中。至此，一系列的分布式方法的启动处理过程就完成了。

上面，详细介绍了分布式方法的调用过程，正确地理解这一调用过程，有助于对后续章节的学习，特别是有助于对客户端和服务器程序设计过程的理解。

第3章 分布式对象系统设计与IDL定义

同一般的面向对象系统的开发一样，分布式对象系统的开发也需要首先从系统的设计开始。系统设计的结果将为系统的实现提供依据。同一般面向对象系统不同的是，分布式对象系统的设计还包括对IDL接口的设计。IDL接口的设计为分别独立地实现客户端程序和服务器程序打下基础。本章主要介绍分布式对象系统的设计过程以及IDL语言的基本语法构成和定义方法。

3.1 分布式对象系统的开发流程

利用CORBA规范来开发分布式对象系统的基本流程如图3.1所示。

由图3.1可知，分布式对象系统的开发流程是由系统设计、IDL接口定义和语言映射、客户端开发、服务器开发以及总体测试与运行等几部分组成的。

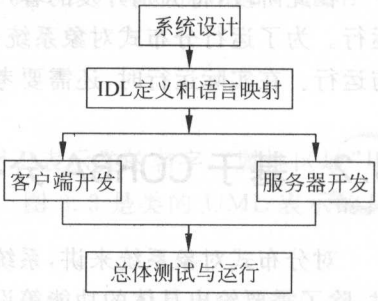

图3.1 分布式对象系统的开发流程

1. 系统设计

基于CORBA系统开发的第一步是系统设计。一般来讲，分布式对象系统的设计同本地对象系统的设计是一样的。

面向对象程序设计的结果(对象的表述及对象的联系等)，可以利用统一建模语言(Unified Modeling Language，UML)来描述。

2. IDL定义和语言映射

从这里开始是CORBA特有的部分。在上述系统设计的结果当中，作为分布式对象实现的对象接口，需要利用IDL来进行描述。这通常是利用手工操作来完成的，但是，如果系统设计的结果是利用UML来表述的话，也可以自动生成(即自动将UML表述转换成相应的IDL定义)。

不管怎样，当IDL接口描述结束之后，就需要利用ORB产品提供的IDL编译器，将IDL定义映射为相应的语言，也就是将IDL定义转换为Stub和Skeleton等对于系统开发所必需的Java语言等代码，这一生成过程被称为映射。在CORBA规范中，对每种程序设计语言都有相应的映射规定。这里需要注意的是，不是所有的程序设计语言都可以用来开

发基于CORBA的分布式对象系统的，换句话说，如果能够利用某种语言来开发CORBA应用系统的话，那一定是存在将IDL接口定义映射为那种语言的编译程序。

3. 客户端开发

从这一步开始就进入了实际的程序设计阶段了。一般来讲，客户端开发是在IDL接口映射之后进行的，这是由于在客户端开发过程中需要利用IDL映射后所生成的类，比如Stub类等。这里需要注意的是，客户端与服务器之间的区别只是相对的。一个程序对于某个分布式对象来讲可以是客户端，而对别的分布式对象来讲又可以是服务器。

客户端开发就是通过利用IDL接口中定义的方法，来实现相应的程序功能。

4. 服务器开发

服务器开发主要包括分布式对象实现的定义以及实现CORBA服务器进程的程序开发。同客户端开发相似，服务器开发一般也是在IDL接口映射之后进行的，这是由于在服务器开发过程中需要利用IDL映射后所生成的类，比如Skeleton类等。

在IDL接口定义的支持下，客户端开发和服务器开发可以同时进行，这是由于客户端所要利用的分布式方法以及服务器需要定义的分布式方法的原型定义在IDL接口定义中都确定下来了，这样，只要按照所给出的原型进行调用和定义即可。

5. 综合测试与运行

在此阶段，将分别开发的客户端程序和服务器程序进行综合测试，在此基础上进行实际运行。为了运行分布式对象系统，需要做许多系统设置的准备工作，而不是单纯的设计程序与运行。在实际运行时，还需要考虑系统维护等工作。

3.2 基于CORBA分布式对象系统设计

对分布式对象系统来讲，系统设计是系统开发过程中最重要的阶段。在进行系统设计时，除了需要给出具体的功能等设计之外，还需要考虑如何来表示设计结果的问题。

1. 系统设计的基本步骤

一般来讲，基于CORBA的分布式对象系统设计可以由如下几部分组成。

1）归纳系统的要求，即功能设计

功能设计，即归纳出系统的需求关系。需要根据系统的具体功能要求来设计。有时，在进行功能设计时，还需要给出具体的QoS要求。

2）对象的确定

为了满足上述功能设计的要求，需要确定提供相应服务的对象，这部分的重点并不在于对象的内部构造定义，而在于将哪些对象组合起来，以提供相应的服务。

3）对象的接口设计

所需要的对象确定了以后，就要设计相应对象的接口。在这一阶段，重要的是要明确需要什么样的接口，而不需要考虑接口是怎样实现的。

4）接口的构造设计

每个分布式对象的接口确定了以后，就需要对实现这一接口的构造进行设计。这一步工作的目的就是有利于用 IDL 进行接口定义。如果接口的构造设计结束了，则描述 IDL 定义的信息也就齐全了。

5）对象实现的设计

接口的构造设计结束以后，就需要对实现这一接口的对象实现(object implementation)进行设计，也就是对实现 IDL 接口定义功能的类进行设计。基于这一阶段的设计，CORBA 客户端和服务器的开发基本上就可以进行了。

6）运行环境设计

运行环境设计就是要进行服务器进程的配置等物理方面的设计。由于物理设计直接影响到系统的运行性能，因此，即使系统开发结束以后，也有可能进行改变。

2. 设计结果的表示

软件开发大体上是由三个阶段组成的：需求分析、设计和编码。这三个阶段在面向对象程序设计中分别被称为 OOA(Object Oriented Analysis)、OOD(Object Oriented Design)和 OOP(Object Oriented Programming)。

在上述面向对象程序设计的过程中，特别是在 OOA 和 OOD 的阶段已有很多方法可以利用，这些方法尽管各有差异，但都是在找出对象的静态构造、功能(作用)和动态关系之后，利用相应的描述方法进行描述。目前最为流行的方法是 UML。对于分布式对象的设计结果来讲，也可以利用 UML 来表示。在利用 UML 来形式化表示设计结果的情况下，就有可能自动地将设计结果转化为 IDL 接口定义。

下面以类的描述为例，来说明 UML 的基本用法。

1）类的 UML 表示

类的 UML 表示形式如图 3.2 所示，这里“名字区域”用于表示类的名字，“属性区域”用于表示类的数据成员，“操作区域”用于表示类的成员方法。图 3.3 是类的 UML 表示的具体例子。

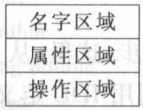

图 3.2　类的 UML 表示

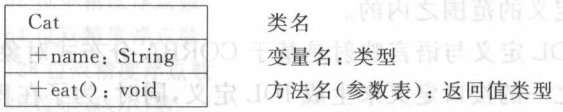

图 3.3　类的 UML 表示的例子

在图 3.3 中，Cat 是类名，name 是数据成员名，冒号后面的部分为该数据成员的类型名。eat 是成员方法名，冒号后面的部分为该函数的返回值类型。在数据成员和成员方法名前面的＋号，表示访问权限。可以设置的访问权限符号及其意义如下：

- ＋：public 权限；
- －：private 权限；
- ＃：protected 权限。

类中的抽象方法可用斜体来表示，而抽象类是在类名的下面用{ }括起来的类的属性来表示的，如图 3.4 所示。

2）类的实例的 UML 表示

类的实例(即对象)的 UML 表示如图 3.5 所示。

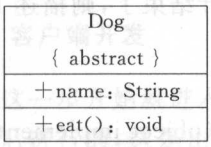

图 3.4 抽象类的表示

C: Cat	对象名：类名
name: "Nothing"	对象中的变量名：值

图 3.5 类的实例的 UML 表示例子

在图 3.5 中，C 是 Cat 类的对象，该对象中的 name 数据成员的值为字符串 Nothing。

3）继承的 UML 表示

继承是面向对象程序设计中的重要概念，在 UML 中也提供了有关继承的表示方法。图 3.6 是 UML 表示继承的一个例子。

图 3.6 的 UML 图说明了 Mammal 是基类，Dog 和 Cat 都是通过继承 Mammal 类而定义的派生类。

UML 的基本内容很多，感兴趣的读者可以查阅相关书籍。总之，分布式对象的设计结果是可以利用 UML 来进行描述的。

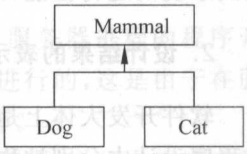

图 3.6 继承的 UML 表示

3.3 IDL 接口定义语言

IDL(Interface Definition Language，接口定义语言)是用于描述分布式对象接口的定义语言，利用 IDL 进行接口定义之后，就确定了客户端与服务器之间的接口，这样即使客户端和服务器独立进行开发，也能够正确地定义和调用所需要的分布式方法。

3.3.1 IDL 的作用

IDL 是 CORBA 的重要的组成部分，这是由于 CORBA 分布式对象能够提供怎样的接口，完全是由 IDL 来规定的。换句话说，CORBA 能够实现的分布式对象接口，是在 IDL 所能够定义的范围之内的。

IDL 定义与语言映射是基于 CORBA 分布式对象系统开发的第二步，这里，从系统设计结果之一的接口定义来生成 IDL 定义，同时，为了在程序设计中能够利用 IDL 定义，需要进行语言映射。在具体介绍 IDL 定义与语言映射之前，先看一看 IDL 的作用。

IDL 的作用就是使分布式对象的接口能够独立于程序设计语言来进行描述。在能够处理各种程序设计语言的系统环境当中，不依赖于程序设计语言而能够进行接口描述是重要的功能。

为了使 IDL 定义的接口能够被实际的程序设计语言所接受，就必须将其转换为相应的程序设计语言的代码，这一过程被称为语言映射，实现这一功能的工具被称为 IDL 编译器。

同一个 IDL 接口，经过不同的语言编译器编译之后，就能够被映射为不同语言所需要的类，这样，就可以采用不同的语言来分别实现服务器和客户端，也就是说客户端和服务器不一定需要采用同一种语言来实现。

我们知道，IDL 是用于定义 CORBA 分布式对象的接口的，IDL 接口的定义是在客户端

程序设计和服务器程序设计之前完成的。目前 IDL 定义还是通过手工编码来实现的，也许在不远的将来，能够从 UML 的设计数据自动生成 IDL 接口定义。但不管怎样，掌握 IDL 的语法构成都是必须的。

为了定义 CORBA 对象接口，IDL 提供了 7 种形式的类型定义，同时还提供了多种数据类型。由 IDL 提供的 7 种定义是：类型定义、常量定义、异常定义、属性定义、操作定义、接口定义和模块定义。

下面具体介绍 IDL 语法及其功能。

3.3.2　数据类型

CORBA 版本不同能够使用的 IDL 语言中的数据类型也不相同。在 CORBA 2.0 及其以后的版本中，能够使用的 IDL 中的数据类型包括对象引用类型、基本类型以及复合类型等。如表 3.1～表 3.3 所示。

1. 对象引用类型

IDL 对象引用类型如表 3.1 所示，其类型名为 Object。

表 3.1　对象引用类型

类型名	说　明
Object	指向 CORBA 对象的指针

2. 基本数据类型

IDL 提供了大量的基本数据类型，如表 3.2 所示。

表 3.2　基本数据类型

基本数据类型	说　明
short	16 位有符号整数
long	32 位有符号整数
long long	64 位有符号整数
unsigned short	16 位无符号整数
unsigned long	32 位无符号整数
unsigned long long	64 位无符号整数
float	32 位单精度浮点数
double	64 位双精度浮点数
long double	128 位双精度浮点数
fixed	32 位定点(小)数
char	1 字节字符
wchar	支持多字节代码集的字符数据类型
string	字符串
wstring	多字节字符串
boolean	布尔值：TRUE 或 FALSE
octet	在传递中不发生变换的 8 位数据(如传递二进制数)
any	可以存放任意类型的数据

3. 复合数据类型

复合数据类型如表 3.3 所示。

表 3.3 复合数据类型

类型名	说　明	类型名	说　明
enum	枚举	union	联合(共用体)
sequence	任意类型的一维数组	数组[ ]	任意类型的任意维数组
struct	结构(结构体)		

在 IDL 定义中，对变量名以及操作名(方法名)等标识符的命名规则如下：

(1) 标识符是由英文字母(a～z，A～Z)、数字(0～9)以及下划线(_)组成的，且标识符的第一个字符必须是英文字母。

由上述命名规则可知，下面的标识符名是合法的：

```
isEmpty,intfol,MAX_SIZE
```

而下面的标识符名是不合法的：

```
10Base,_Data,$STR
```

(2) IDL 定义中的大写字母和小写字母是有区别的，即作为不同的字母来看待。如标识符 find 和 Find 将分别表示两个不同的标识符。

(3) 关键字不能作为一般用户定义的标识符。

下面将具体介绍在 IDL 中可以利用的 7 种类型的定义格式及其作用。

3.3.3 类型定义

类型定义主要包括 typedef、struct、union、enum 和 sequence。

1. typedef 定义

typedef 的定义格式如下：

```
typedef 已有类型名　别名;
```

例如：

```
typedef  string  Name;              //可以利用 Name 来定义 string 类型的变量
typedef  long  Matrix[10][10];      //可以利用 Matrix 来定义 10×10 二维 long 类型数组
```

在程序中使用 typedef 的主要目的是为了提高程序的可读性。

2. struct 定义

struct 是用于定义结构类型的关键字，其基本用法如下：

```
struct 结构名
{
    类型名  变量名;
    类型名  变量名;
    ⋮
};
```

例如：

```
struct  sPoint
{
    long  x;
    long  y;
};
```

其中，sPoint 为结构类型名，该结构包含两个 long 类型的数据成员。

3. union 定义

union 是用于定义联合类型的关键字，其基本用法如下：

```
union  联合名  switch(区分符的类型名)
{
      case 区分符的值 1:类型名  变量名;
      case 区分符的值 2:类型名  变量名;
      ⋮
      default:类型名  变量名;
};
```

例如：

```
union uItem switch(short)
{
    case 1:long lval;
    case 2:double dval;
    default:string sval;
};
```

其中，uItem 是联合类型名，switch 后面括号中的 short 表示该联合类型是由短整型值区分的，如果该短整型的值为 1，则利用 lval 成员来存储 long 类型数据，如果该短整型的值为 2，则利用 dval 成员来存储 double 类型数据，如果该短整型的值既不是 1 也不是 2，则利用 sval 成员来存储 string 类型数据。

需要注意的是，switch 后面括号中的“区分符的类型名”只能是 integer、char、boolean 和 enum 等类型，同时，case 后面的值的类型必须与“区分符的类型名”相匹配。例如：

```
enum EnumType{a,b,c,d};
union UnionType switch(EnumType)
{
    case a:long field_a;
    case b:string field_b;
    default:boolean field_z;
};
```

4. enum 定义

enum 是用来定义枚举类型的关键字，其基本用法如下：

```
enum  枚举类型名 {name1,name2,…,nameN};
```

例如：

```
enum CalcMode{HEX,OCT,BIN,DEC};
```

在此定义的基础上，如果定义了 CalcMode 类型变量，则该变量可以取 HEX、OCT、BIN 和 DEC 中的任一个值。例如：

```
enum ABCDType{A,B,C,D};
union UType switch(ABCDType)
{
    case A:long AType;
    case B:string BType;
    case C:double CType
    default:boolean DType;
};
```

5. sequence 定义

sequence 用于定义任意类型的一维数组。其基本用法如下：

```
sequence<类型名>变量名;
```

例如：

```
sequence<short>sv;
```

它表示 sv 是一个无界的短整型一维数组（或称一维序列）。如果希望限定一维序列的上限，则可以采用 sequence<类型名,上限>的方式来定义。例如：

```
sequence<short,100>sx;
```

它表示 sx 是一个上限不超过 100 的短整型一维序列。

需要注意的是，类型定义中的很多定义与 C 语言中的相应定义是不同的，有的理解起来还有一定的难度，但这里给出的只是形式上的定义而已，这里的形式定义只能在 IDL 接口定义中直接使用，而真正在程序中需要使用的则是其映射后的结果。有关类型定义的映射请参考本章的后续内容。

3.3.4 常量定义

IDL 常量定义的格式如下：

```
const 类型名 常量名 = 常量表达式;
```

例如：

```
const long MAXSIZE = 100;
```

利用这种方式定义的常量，不但在接口定义中可以引用，而且在实现接口的程序中也可以引用。能够定义常量的数据类型有整型、浮点型、定点小数类型、字符类型、字符串类型和布尔类型等。

常量表达式不仅仅可以是一个常数，而且可以是包括运算符的常量表达式。在常量表

达式中可以包含如下一些运算符：

- 位运算符：|（按位或）、&（按位与）、^（按位异或）、~（按位取反）；
- 移位运算符：>>（右移）、<<（左移）；
- 加减乘除运算符：+（加法）、-（减法）、*（乘法）、/（除法）、%（求余）；
- 单项运算符：+（正）、-（负）。

3.3.5　异常定义

IDL 异常定义的格式如下：

```
exception  异常名
{
    类型名 1   变量名 1;
    类型名 2   变量名 2;
       ⋮
};
```

在实际应用过程中，一般在异常类中不定义变量或只定义一个变量。例如：

```
exception ServerException { };
exception NotFoundException { string module_name;};
```

在 ORB 中，当因某种原因调用函数（或称操作）失败时，为了识别发生了何种异常，需要将必要的信息存入特定的数据结构（异常中定义的数据结构）中，并将此数据结构传递给调用此函数的程序中。换句话说，在调用分布式方法失败时，需要将异常中定义的数据结构返回给客户端程序。

需要注意的是，在 CORBA 中不支持异常的继承操作。

CORBA 中规定的异常有两种：系统异常和用户异常。系统异常的入口由 ORB 全部规定好，当发生系统异常时，被传递的数据结构也都确定好了。用户异常是异常名和数据结构都需要由用户来定义的异常，其定义的形式如前所述。

如上面的例子那样，异常定义中可以为空。

在调用哪个操作时发生用户定义的异常，将在后面叙述。

3.3.6　属性定义

服务器向客户端公开的属性是利用属性定义来描述的。

属性定义的格式如下：

```
[readonly] attribute   类型名   属性名;
```

例如：

```
attribute long count;
```

如果我们把 IDL 定义中的操作看作为 Java 类中的公有方法的话，那么属性就可以被看作是公有的变量。

作为属性定义的数据可以被客户端所访问（包括读和写），但是，如果在定义属性的前面

加有 readonly 关键字，则只能对该属性进行读操作。属性定义只能在后面将要叙述的 interface 定义中使用。

在对属性进行访问时，也可能发生标准异常。

属性的基本概念可通过对下面两个等价的模块定义的比较来理解。

下面的模块定义是比较原始的，仅仅是为了说明问题而已，一般在实际运用中并不这样定义。

```
module common
{
    interface commonItem
    {
        string get_name();
        void set_name(in string name);
        long get_amount();
        void set_amount(in long amount);
        double get_price();
        void set_price(in double price);
    };
    ⋮
};
```

从上面的接口定义可知，非常烦琐，接口中的 6 个操作定义仅仅能够读/写对象的三个状态变量，而且 6 个定义可以分为 3 组，每组之间都非常相似。显然，这种定义方式非常烦琐。

考虑到读取、修改对象状态变量是一个相当普遍的操作，在 IDL 中引入了属性的概念，用 attribute 定义。利用 attribute 重新定义上述接口如下：

```
module common
{
    interface commonItem
    {
        attribute string name;
        attribute long amount;
        attribute double price;
    };
    ⋮
};
```

显然，后一种定义方式比前一种定义方式更简单明了。

3.3.7 操作定义

操作相当于 Java 类中的公有方法，其定义格式如下：

```
[oneway] 返回值类型  操作名(参数 1,参数 2,…)
                [ raises(异常名 1,异常名 2,…)]
                [context("字符串 1","字符串 2",…)];
```

例如：

```
string getDocument(in string url)
        raises(NotFound)context("HTML_VERSION");
```

操作的定义是同来自于客户端的调用请求相对应的。在操作的定义中包含操作名(函数名)、数据的传递列表(参数表)和返回值类型。

CORBA 的操作有两种，它是由在操作定义的前面是否加有 oneway 来决定的。oneway 是用于对操作的属性进行定义的。如果在操作定义的前面没有加上 oneway，则在调用操作时，操作执行的成功与否可以在调用方进行确认，即发生异常时失败，其余情况成功；如果在操作定义的前面加上 oneway，则只能保证调用请求信息的发送，而以后的执行情况就没有任何保证了。换句话说，对于不加 oneway 的操作来讲，直到返回执行结果都一直处于同步等待状态；而对于加有 oneway 的操作来讲，当分布式对象的请求发出以后，可以立即转入下一个处理，而不等待操作的执行。由此可知，oneway 操作不能有返回值，同时 oneway 操作中的参数属性只能是 in，也就是只能从客户端向服务器传递参数，而不能从服务器向客户端传递数据。

操作定义中的参数说明格式如下：

```
in | out | inout   类型名   参数名
```

例如：

```
in string url
```

这里 in、out 和 inout 是参数属性，它用于指定参数的传递方向。同时只能指定一个参数属性。

表 3.4 给出了 in、out 和 inout 参数属性的意义。

表 3.4　操作中参数属性的意义

in	从客户端向 CORBA 对象传递参数
	在调用操作之前，在客户端设置参数区域，在客户端设置参数值。在调用操作时，复制参数值，并将复制的参数值由客户端传递给 CORBA 对象。参数可以是常量，也可以是变量
out	从 CORBA 对象向客户端传递数据
	在操作的处理过程中，在 CORBA 对象中设置参数区域，在 CORBA 对象中设置参数值。操作处理结束以后，复制参数值，并将复制的参数值由 CORBA 对象传递给客户端。参数只能是变量
inout	在客户端和 CORBA 对象之间传递参数和数据
	在调用操作之前，在客户端设置参数区域，在客户端设置参数值。在调用操作时，参数的引用(指针)被从客户端传递给 CORBA 对象。在 CORBA 对象中，在操作的处理过程中，通过使用被传递过来的参数引用，来读取和更新客户端中的参数值。参数只能是变量

在定义操作时参数的属性必须要指定。

操作定义中的 raises，用于列出所有在调用操作过程中可能发生的异常。用 raises 能够列出的异常名只限于事先利用前述的异常定义进行描述的异常。

操作定义中的 context 不是作为参数使用的，它是从客户端向服务器传递数据的手段。

通过指定 context 后面的 context 名，就可以在发出调用请求的同时，来决定所要传递的内容(context)。一般来讲，可以利用 context 来传递 ORB 执行平台的环境变量等信息。

例如，可以在调用操作时利用 context 来指定所要传递的信息的种类（如，LANGUAGE,POSTSCRIPT_VERSION,JDK_VERSION)以及 CORBA 对象所连接的网络能够使用的资源(PRINTER,SCANNER)等。

采用 context 上下文对象来传递非参数信息时，该信息仅在一段时间内有效，而且客户端必须通过 ORB 提供的方法来处理上下文对象。服务器可以根据要求来读取上下文信息，以决定自己的行为方式。同时，上下文对象的取值必须为 string 类型。由 context 所指定的内容，相当于环境变量，它表示要在给定的方法中利用此环境变量的值，其真正的值要在客户端或服务器进行设置。

在客户端调用带有 context 的方法时，需要同时将所设定的 context 变量名及其值也传递给服务器，以便在服务器方进行使用。客户端在设置 context 变量及其值时需要利用系统定义的 context object 对象。

3.3.8 接口定义

IDL 接口定义的格式如下：

```
interface  接口名[:继承的接口名 1,接口名 2,…]
{
    (1) 类型定义
    (2) 常量定义
    (3) 异常定义
    (4) 属性定义
    (5) 操作定义
};
```

例如：

```
interface MatrixCalc
{
    const long SIZE_X = 100;
    const long SIZE_Y = 100;
    typedef long Matrix[SIZE_X][SIZE_Y];
    Matrix add(in Matrix m1,in Matrix m2);
    Matrix sub(in Matrix m1,in Matrix m2);
};
```

说明：

(1) 接口定义是利用前述的类型定义、常量定义、异常定义、属性定义和操作定义等来描述 CORBA 对象的接口。尽管在使用这些定义的顺序上没有什么要求，但对于后面使用的类型定义以及接口等必须在使用前进行定义。

(2) 一个接口定义与 CORBA 对象的一个接口相对应。

(3) 在接口定义中可以继承已有的接口。为了继承已有的接口，可直接在冒号的后面列出要继承的接口名。通过继承而生成的接口中将包含被继承的接口中的所有内容(类型、常量、异常、属性和操作等)。操作的说明不支持多态性(这里的多态性是指具有不同参数个

数和类型的同名操作可以定义多个)。

3.3.9 模块定义

IDL模块定义的格式如下:

```
module 模块名
{
    (1) 类型说明
    (2) 常量说明
    (3) 异常说明
    (4) 接口说明
    (5) 模块说明
};
```

在interface定义中,如果定义的内容很多,则会使各定义之间的关系变得不明确,同时,也容易使各定义中使用的名字出现重复。

模块(module)定义将相关的多个定义放在一个模块中(module),这样,一个模块就可以形成一个有效范围。

如果利用module定义,就可以在下面两个模块中定义同名的类型xxx和同名的接口yyy,也就是说不同模块中的类型定义和接口定义等可以重名。

```
module MA
{
  typedef long xxx;
  interface yyy {…};
};
module MB
{
  typedef sequence<long>xxx;
  interface yyy {…};
};
```

在使用模块定义时,本模块内的标识符可直接引用(这就是所谓的命名作用域问题),但其他模块中的标识符不能直接引用。为了引用其他模块中的标识符,就需要在标识符的前面加上"::"以及相应的模块名。例如:

```
module MA
{
   typedef long xxx;
   interface yyy {…};
};
module MC
{
   struct PPP
   {
       string name;
       MA::xxx val;
   };
   interface zzz:MA::yyy
```

```
    {
        ⋮
    };
};
```

在上述定义中，作为 PPP 结构成员的 val 是利用 MA 模块中定义的 xxx 类型来定义的，而接口 zzz 则是通过继承 MA 模块中的 yyy 接口生成的。

在模块定义的内部还可以包含模块定义，以形成模块的层次结构，这时，可通过“::”来进行引用。例如：

```
module MA1
{
  module MA2
  {
     module MA3
     {
        typedef string RRR;
     };
  };
};
module MD
{
   typedef MA1::MA2::MA3::RRR DDD;
};
```

这里，模块 MD 中的类型 DDD 是利用 MA1::MA2::MA3 模块中的 RRR 类型来定义的。

3.3.10 预处理器

IDL 的预处理器同 C++的预处理器具有等价的功能，其主要内容如表 3.5 所示。

表 3.5 IDL 预处理器

功　能	举　例
注释(/*…*/,//)	/* This is comment */ String name; //length<10
宏定义(#define, #undef)	#define MaxSIZE 100 #define SEQ(type) sequence<type>
条件编译(#if, #ifdef, #ifndef, #else, #endif)	#ifndef BASIC_H #define BASIC _H #endif
文件包括(#include)	#include“comment. idl”
行号控制(#line)	#line 100
依赖于实现的动作(#pragma)	#pragma iiop

由表 3.5 可知，IDL 预处理器与 C++语言中的预处理器在功能上基本是一样的，下面仅对#line 和#pragma 进行简单说明。

1. #line 行号

#line 预处理指令表示设定该行所在的下一行的行号。例如，假如在接口定义中有下面两条语句：

```
#line 100
const short a = 200;
```

它表示第二条语句所在行号为 100。

设置#line 预处理指令的主要目的是用于确定当发生错误时该错误所在的行号。

2. #pragma 操作

#pragma 指令用于在程序中向编译器发出指示，以便使编译器能够进行相应的处理。一般向编译器发出指令是在命令行上进行的，而#pragma 则可以在程序中向编译器发送指令。具体能够发送何种指令，完全是由编译器决定的。

3.4　从 IDL 到 Java 的映射

CORBA 对象可以利用各种语言来实现，说某种语言能够实现 CORBA 对象是指存在从 IDL 到那种语言的映射。例如，在 CORBA 2.2 中，就规定了从 IDL 到 C、C++和 Java 等语言的映射规则。只有将 IDL 定义的接口信息映射为相应的语言之后，才能在程序中利用这种语言来进行分布式对象系统的设计。本节主要介绍从 IDL 到 Java 语言的映射过程，由于即使映射为相同的语言，但如果 CORBA 系统不同其映射结果也可能是不一样的，因此，这里只是给出一种可能的映射结果。

3.4.1　接口定义的映射

在 IDL 到 Java 的映射过程中，一个 IDL 接口(interface)定义被映射为一个 Java 语言的接口以及 4 个主要的类。

下面以下述的 HelloServer 的 IDL 接口定义为例来说明 IDL 编译器的映射结果。

```
interface HelloServer
{
    string sayHello(in string name);
};
```

HelloServer 接口经 IDL 编译器映射为 Java 语言之后，所生成的 Java 语言程序主要是由下述五个部分组成，其中的 Java 语言接口以及各类的名字是以 IDL 接口的名字为基础来命名的。下面给出每一部分的映射结果并说明其具体作用。

1. Java 语言接口

IDL 接口定义中的 interface 首先被映射为 Java 语言中的 interface，其映射结果如下：

```
public interface HelloServer extends org.omg.CORBA.Object
```

```
{
    String sayHello(java.lang.String name);
}
```

其中，org.omg.CORBA.Object 定义了所有 CORBA 对象的通用方法，以后将具体介绍。

作为映射结果的 Java 语言接口，是需要在下面介绍的 stub 类和 skeleton 类中进行实现的接口。stub(代理对象)是通过该接口被客户端所调用的，skeleton 则是通过具有该接口的实现程序来调用每个操作的。该接口的定义必须要继承 org.omg.CORBA.Object 接口。在 org.omg.CORBA.Object 接口中，定义了 CORBA 对象应该具有的功能，其中包括对象引用的复制、比较以及用于生成 Request 类的实例等函数。org.omg.CORBA.Object 接口的实现是由各厂家提供的，用户不需要实现。

2. skeleton 类

skeleton 类的定义如下：

```
public abstract class  _HelloServerImplBase
                       extends org.omg.CORBA.DynamicImplementation
                       implements HelloServer
{
    ⋮
}
```

其中，_HelloServerImplBase 是 skeleton 类名，它是随着系统的不同而不同的。skeleton 类中的具体内容将在后续的章节中进行介绍。

skeleton 类的作用是作为分布式对象实现的基类。由上述定义可知，skeleton 类本身是由 IDL 编译器产生的抽象类。skeleton 类的父类是随厂家的不同而不同的，在这里其父类是 org.omg.CORBA.DynamicImplementation。

skeleton 类的父类主要完成将发送给 CORBA 对象的请求信息传递给 skeleton 类。

由上述 skeleton 类的定义不难看出，作为映射结果的 Java 语言的接口 HelloServer，在 skeleton 类中要进行实现。

3. stub 类

stub 类的定义如下：

```
public class  _HelloServerStub
              extends org.omg.CORBA.portable.ObjectImpl
              implements HelloServer
{
    ⋮
    String sayHello(java.lang.String name){…}
    ⋮
}
```

其中，_HelloServerStub 是 stub 类名，它是随着系统的不同而不同的。stub 类中的具体内容将在后续的章节中进行介绍。

stub 类是作为客户端程序的代理对象的类由 IDL 编译器生成的。stub 类必须要继承

org. omg. CORBA. portable. ObjectImpl 类。org. omg. CORBA. portable. ObjectImpl 类主要完成从 stub 类中取出请求信息并将其传递到 CORBA 对象中的任务。

由 stub 类的定义不难看出，作为映射结果的 Java 语言接口 HelloServer，要在 stub 类中给出其定义。由于 stub 是客户端的代理对象，因此，在客户端调用分布式对象时并不是直接调用服务器上的分布式对象，而是调用 stub 类中的同名方法，在该方法中来完成分布式方法的调用过程。这样，从程序设计的角度来看，对分布式方法的调用就如同调用本地方法一样的简单，许多复杂的调用细节都被 stub 类屏蔽掉了。

4. Holder 类

Holder 类是与 IDL 数据类型相对应的 Java 语言的类，即每个 Holder 类中都用于保存着一种类型的数据。每一个接口定义、每一个基本数据类型以及每一个用户自定义类型都有一个 Holder 类与其对应。

在利用 IDL 来定义接口信息时，可以指定操作的每个参数的传递方向（in、out 和 inout），这种处理方式与 Java 语言中的参数处理方式是不同的，下面对参数的处理方式进行简单的说明。

1) in 参数

如果参数的传递方向被指定为 in，则表示只是从客户端向服务器传递参数。在服务器中的参数修改不会被传递给客户端。当将 IDL 操作中的 in 参数映射为 Java 语言中的基本数据类型时，正好能够满足这一要求，这是因为以 Java 语言中的基本数据类型作为参数所采用的就是传值的方式，也就是只能从客户端向服务器传递参数。

2) out 参数

如果参数的传递方向被指定为 out，则只能从服务器向客户端传递参数值，客户端不能向服务器传递参数，服务器也不能取参数。参数的这种传递方式利用 Java 语言中的参数处理方式是无法实现的。为此，为了实现 out 参数传递，需要利用 Holder 类这样的间接方式来实现。

在所有的 Holder 类中，都定义了保存对应类型数据的公有的数据成员和两个构造函数，其中一个构造函数是不为数据成员设置任何初值的默认构造函数，另一个构造函数是具有参数的、用于为数据成员设置初值的。

对于 out 参数来讲，一般是利用默认构造函数来生成对应参数类型的 Holder 类的对象，并作为参数传递给服务器。这时，在客户端的 ORB 中不处理 out 参数中的值，在来自客户端的请求信息中不包含 out 参数（尽管不包含 out 参数，但相应的 Holder 类对象还必须要创建，这是由于在从服务器返回参数值时需要利用该 Holder 类）。

在服务器端的 ORB 中，利用默认构造函数来生成不保存数据的 Holader 类，并传递给分布式对象实现，在分布式对象实现中将处理结果保存到 Holder 对象中。在服务器端的分布式方法执行完之后，在服务器端的 ORB 中取出保存在 Holder 对象中的参数值并生成回答信息（reply message）后发送给客户端。在客户端的方法调用返回之后，在客户端程序中通过访问 Holder 类中的数据成员，就可以使用从服务器返回的参数数据。

3) inout 参数

如果参数的传递方向被指定为 inout，则既从客户端向服务器传递参数，同时在分布式

方法执行完了之后，也通过该参数向客户端传递数据。在 Java 映射中，这种方式的参数传递需要利用 Holder 类这样的间接方式来实现。

在 inout 参数处理中，为了从客户端向服务器传递数据，可以利用 Holder 类中能够指定初值的带参数的构造函数来生成 Holder 类的对象，并以该 Holder 对象为参数来调用分布式方法。在客户端的 ORB 中，通过利用 Holder 类中的数据来生成请求信息（request message）并将其发送给服务器。

服务器端的 ORB 从请求信息中取出数据，生成保存该数据的 Holder 对象并传递个分布式对象实现。在分布式对象实现中，从 Holder 对象中取出数据并利用之，同时将返回给客户端的参数值保存到 Holder 对象中。

在服务器端的分布式方法执行完之后，在服务器端的 ORB 中取出保存在 Holder 对象中的参数值并生成回答信息（reply message）后发送给客户端。在客户端的方法调用返回之后，在客户端程序中通过访问 Holder 类中的数据成员，就可以使用从服务器返回的参数数据了，这一过程与 out 参数处理完全一样。

上述 IDL 接口定义被映射为 Java 语言中的 Holder 类的形式如下：

```
public final class HelloServerHolder
{
    ⋮
}
```

在 IDL 接口定义中描述的接口定义、用户自定义类型以及 IDL 可用的基本类型等都有一一对应的 Holder 类定义。

基本类型的 Holder 类在 CORBA 中已定义好，需要时直接利用即可。用户定义类型的 Holder 类在 IDL 编译时产生。有关 Holder 类的具体内容将在后续章节中详细叙述。

5. Helper 类

IDL 接口定义被映射为 Java 语言中的 Helper 类的形式如下：

```
public final class HelloServerHelper
{
    ⋮
}
```

作为映射结果的 Helper 类用于提供各种实用功能。例如，对 any 类型数据的操作以及 narrow 方法等。在 Java 映射中，在 IDL 接口定义中描述的接口定义和用户定义类型都有一一对应的 Helper 类映射结果。有关 Helper 类的详细说明请见后续章节。

3.4.2 实现引用传递的 Holder 类

在利用 IDL 编译器来实现服务器和客户端程序时，开发者可以直接与 Holder 类打交道。因此，在本节再详细介绍一下 Holder 类。

如前所述，Holder 类是为了在调用接口中提供的方法时实现引用传递而定义的类。所谓引用传递，是指不是单纯的传递数据，而是将该数据的地址引用也传递过去。这种参数传递方式在 C/C++语言中可通过指针来实现。

需要进行引用传递的接口定义的例子如下：

```
interface HelloServerByRef
{
    void sayHello(in string name,out string reply);
};
```

在此定义接口中，sayHello 函数的返回值为 void，参数 name 的属性为 in，而参数 reply 的属性则为 out 类型。

out 类型的参数用于将数据从 CORBA 对象传递到客户端。在各种语言的映射中，可直接利用语言中“引用传递”的功能来实现 out 类型的处理。同时，inout 类型的参数也具有引用传递功能。

由于 Java 中的基本类型数据作参数时只能传值，因此在映射的时候，就不能直接利用 Java 中的基本类型数据来进行 out 和 inout 参数传递。

如前面所述，在 CORBA 中是利用 Holder 类来处理 out 和 inout 参数的，这本身也是利用 Java 语言中对象作为函数参数是采用引用传递的特点，当然，由于客户端和服务器不在一个进程中，采用 Holder 类来处理 out 和 inout 参数是属于一种间接的处理方式，与同一个进程中的引用传递是不同的。上述 HelloServerByRef 接口中的 sayHello 方法可映射为如下的 Java 方法：

```
void sayHello(java.lang.String name,CORBA.StringHolder reply)
{
    ⋮
}
```

从上述映射结果不难看出，IDL 语言中的 in 属性的 string 类型被映射为 Java 语言中的 java.lang.String 类型，而 out 属性的 string 类型则被映射为 CORBA.StringHolder 类。

在 IDL 到 Java 语言的映射处理中，如果在方法中指定了 out/inout 类型的参数，则此时不是将其直接映射为 Java 语言中的基本数据类型，而是将其映射为那种类型的 Holder 类。

由于在程序设计中可以直接使用 Holder 类，因此下面给出几个 Holder 类的例子。

1. long 类型的 Holder 类

由于 IDL 中的 long 类型与 Java 语言中的 int 类型等价（都是 32 位），因此属性为 out 和 inout 的 IDL 中的 long 类型参数将被映射为 Java 语言中的 IntHolder 类，该类的定义如下：

```
final public class IntHolder
{
    public int value;
    public IntHolder(){ }
    public IntHolder(int initial)
    {
        value = initial;
    }
```

```
}
```

由于所有成员都是 public 的(包括数据成员)，因此，在任何函数中都可以利用“对象名. value”取出 value 的值。

2. string 类型的 Holder 类

属性为 out 和 inout 的 IDL 中的 string 类型参数将被映射为 Java 语言中的 StringHolder 类，该类的定义如下：

```
final public class StringHolder
{
    public java.lang.String value;
    public StringHolder(){ }
    public StringHolder(java.lang.String initial)
    {
        value = initial;
    }
}
```

3. 接口以及用户定义类型的 Holder 类的基本形式

下面的程序段给出了接口及用户定义类型的 Holder 类的基本形式。

```
final public class xxxxHolder
                implements org.omg.CORBA.portable.Streamable
{
    public xxxx value;                                    //相应类型的变量
    public xxxxHolder(){ }
    public xxxxHolder(xxxx initial){ value = initial;}
    public void _read(org.omg.CORBA.portable.InputStream i)
    {…}                                                   //从“流”中读数据到 value 中
    public void _write(org.omg.CORBA.portable.OutputStream o)
    {…}                                                   //将 value 数据写入“流”中
    public org.omg.CORBA.TypeCode _type(){…}              //返回 xxxx 类型的 TypeCode
}
```

其中，xxxx 表示接口以及用户定义类型的标识符。

前已述及，在 IDL 到 Java 的映射过程中，不仅仅映射到 Java 语言的基本数据类型，对于在 IDL 中能够使用的所有数据类型(包括用户定义类型)来讲，被指定为 out/inout 属性的参数的传递处理都是通过使用 Holder 类来实现的。

3.4.3 提供各种实用功能的 Helper 类

Helper 类也是由 IDL 编译器产生的，程序设计者可以直接利用其功能。不但接口定义，像结构、联合这样的用户定义类型也被映射为相应的 Helper 类，其中提供了利用接口及用户定义类型的功能，但绝大部分方法都是由 stub 和 skeleton 类使用的，只有用于 Any 类型操作的方法以及 narrow 方法等，在程序设计时才能使用到。

Helper 类提供了使用接口及用户定义类型来编写程序时所需要的实用方法，其定义的

基本形式如下。

```
public class xxxxHelper
{
```

(1) 用于 Any 类型处理。

```
public static void insert(org.omg.CORBA.Any a,xxxx t)
{…}     //将 xxxx 类型数据插入到 Any 类型变量中
public static xxxx extract(Any a)
{…}     //从 Any 类型中抽取出 xxxx 类型数据
```

(2) 用于取出类型码(Typecode)。

```
public static org.omg.CORBA.TypeCode type()
{…}     //返回 xxxx 类型的 TypeCode 对象
```

(3) 用于取出 RepositoryID。

```
public static string id()
{…}
```

(4) 用于序列化(serialize)。

```
public static xxxx read(org.omg.CORBA.portable.InputStream istream)
{…}
public static void write(org.omg.CORBA.portable.OutputStream ostream,xxxx value)
{…}
```

(5) 下面的方法仅在有 interface 定义的时候存在。

```
    public static xxxx narrow(org.omg.CORBA.Object obj)
    {…}
}
```

其中,xxxx 表示接口以及用户定义类型的标识符。

下面根据程序中的序号对上述 Helper 类的定义进行简单的说明。

(1) 用于将 xxxx 类型的对象或者接口插入 Any 类型的对象中,或者从 Any 对象中取出 xxxx 类型的对象或接口。

(2) 由 IDL 定义(描述)的类型,都具有一个被称为类型码(TypeCode)的字符串类型名,此方法就是为了取出类型码的。

(3) 用于取出 xxxx 的类型定义或者接口定义在接口仓库中的 ID,即 Repository ID。

(4) 用于将 xxxx 类型的对象或接口变换成“流式”数据,即序列化处理。

(5) 用于生成代理对象,并将 CORBA 对象的对象引用与其代理对象的引用进行关联处理。也就是根据由服务器传递过来的对象引用来生成相应 stub 类的代理对象。

Helper 类中的 narrow 方法在程序设计过程中是经常被使用的方法,该方法的具体意义在前面的章节中已经进行了详细说明。

3.4.4　其他 IDL 定义的映射

1. 模块定义的映射

IDL 中的模块定义(module)是为了形成名字的作用域而使用的定义。在 IDL 到 Java

的映射中，模块定义被映射为 Java 语言中用于形成名字作用域的包(package)。表 3.6 给出了 IDL 模块定义的一种可能的映射结果。

表 3.6 IDL 模块定义的 Java 映射结果

IDL 定义	IDL 到 Java 映射结果

module test
{
interface InterTest
{…};
};
	package test;
	public interface InterTest
	extends org. omg. CORBA. Object
	{…}

其中，作为 module 名的 test 被映射为 Java 语言的 package 名。

2. 常量定义的映射

IDL 的常量定义有两种形式，一种是常量直接在模块中定义，另一种是常量在接口中定义。这两种定义的 Java 映射结果是有区别的。表 3.7 给出了 IDL 常量定义的一种可能的映射结果。

表 3.7 IDL 常量定义的 Java 映射结果

IDL 定义	IDL 到 Java 的映射结果

直接在模块中定义的常量
const 类型 标识符＝常量；
	public interface 标识符{
	public static final 映射后的类型
	value＝常量；
	}

在接口中定义的常量
public interface test {
const 类型 标识符＝常量；
};
	public interface test {
	public static final 映射后的类型
	标识符＝常量；
	}

由表 3.7 映射结果可知，不管是直接在模块中定义的常量还是在接口中定义的常量，都被映射为 Java 语言接口中的常量。例如：

```
module test
{
    const string do = "Nothing";
};
```

经 IDL 编译器映射之后，将生成如下 Java 语言代码：

```
package test;
public interface do
{
    public static final java.lang.String value = "Nothing";
}
```

例如：

```
module test
{
    interface contest
    {
        const string do = "Nothing";
    };
};
```

经 IDL 编译器映射之后，将生成如下 Java 语言代码：

```
package test;
public interface contest
{
    public static final java.lang.String do = "Nothing";
}
```

3. 属性定义的映射

IDL 中的属性定义有两种方式，一种是带有 readonly 关键字，另一种是不带该关键字。这两种定义方式的映射结果是不同的。表 3.8 给出了 IDL 属性定义的一种可能的映射结果。

表 3.8　IDL 属性定义的 Java 映射结果

IDL 定义	IDL 到 Java 的映射
attribute 类型 标识符；	映射后的类型　标识符()； void 标识符(映射后的类型 value)；
readonly attribute 类型　标识符；	映射后的类型　标识符()；

由表 3.8 可知，不带 readonly 关键字的属性定义，被映射为两个函数说明，一个具有 get 功能(取出属性值)，一个具有 set 功能(设置属性值)。而带有 readonly 关键字的属性定义，则被映射为一个函数说明，该函数只具有 get 功能(取出属性值)。例如：

假如有如下 IDL 属性定义：

```
attribute long count;
```

该属性定义的 Java 映射结果如下：

```
int count();
void count(int value);
```

又如，假如有如下 IDL 属性定义：

```
readonly attribute long count;
```

该属性定义的 Java 映射结果如下：

```
int count();
```

4. 异常定义的映射

表 3.9 给出了 IDL 异常定义及其可能的映射结果。

表 3.9 IDL 异常定义的 Java 映射结果

IDL 定义	IDL 到 Java 的映射

exception 标识符 {
　　类型 1　变量 1;
　　⋮
　　类型 n　变量 n;
};
	final public class　标识符
	　　extends org. omg. CORBA. UserException {
	　　public 映射后的类型 1　变量 1;
	　　⋮
	　　public 映射后的类型 n　变量 n;
	　　public 标识符()　{…}
	　　public 标识符(
	　　　　映射后的类型 1　变量 1,
	　　　　⋮
	　　　　映射后的类型 n　变量 n)
	　　　　{
	　　　　⋮
	　　　　}
	}
	final public class　标识符 Holder implements
	　　org. omg. CORBA. portable. Streamable
	{…}

由表 3.9 可知，IDL 中的异常定义被映射为 Java 语言中的两个类，一个是异常处理类，另一个则是与异常定义对应的 Holder 类。例如：

```
//假如有如下 IDL 接口定义
module Sample{
    exception UnknownException { };
};
//该 IDL 定义的 Java 映射结果如下
package Sample;
final public class UnknownException
                extends org.omg.CORBA.UserException
{
    public UnknownException(){…}
}
final public class UnknownExceptionHolder
                implements org.omg.CORBA.portable.Streamable
{…}
```

5. 操作定义的映射

IDL 中的操作定义，相当于 Java 语言中的方法定义。表 3.10 给出了 IDL 操作定义及

其可能的映射结果。

表 3.10　IDL 操作定义的 Java 映射结果

IDL 定义	IDL 到 Java 的映射
类型　操作名(typel 类型 1　变量名 1, …,typel 类型 n　变量名 n) raises(异常 1,…,异常 n)	映射后的类型　方法名(映射后的类型 1　变量 1, …,映射后的类型 n　变量 n) throws 映射后的异常 1,…, 映射后的异常 n;

由表 3.10 可知,IDL 中的操作定义,被映射为 Java 语言中的方法,如果有异常抛出的话,也需要进行相应的映射。例如:

假如有如下 IDL 操作定义:

```
long div(in long x,in long y)raises(ZeroException);
```

该 IDL 定义的 Java 映射结果如下:

```
int div(int x,int y)throws ZeroException;
```

又如,假如有如下 IDL 操作定义:

```
oneway void PrintHistory()context("Encoding","File");
```

该 IDL 定义的 Java 映射结果如下:

```
public void PrintHistory(org.omg.CORBA.Context _c);
```

由映射结果可知,如果在操作定义时包含 context 项,则在映射时,系统自动将 context 对象添加到参数列表中,也就是在原有参数的基础上再增加一个参数,以便传递 context 信息。

6. 用户定义类型的映射

在 IDL 接口定义语言中,用户定义类型包括 struct(结构)、union(联合)和 enum(枚举)等。为了叙述方便,先定义如下 IDL 接口:

```
module Test
{
    struct StrType
    {
        long field1;
        string field2;
    };
    union UnionType switch(short)
    {
        case 1:string a;
        case 2:long c;
    };
    enum Day{Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
```

```
    ⋮
};
```

1）结构定义的映射

上面 Test 模块中定义的结构 StrType 的 Java 映射结果如下。

```
final public class StrType
{
    //与结构成员对应的域的定义
    public int field1;
    public java.lang.String field2;
    //构造函数
    public StrType(){ }
    public StrType(int field1,java.lang.String field2)
    {
        this.field1 = field1;
        this.field2 = field2;
    }
    ⋮
}
```

从上面的映射结果不难看出，IDL 中的结构被映射为与其同名的 Java 语言的 final 类。此类是由与结构成员相对应的公共数据成员和构造函数构成的。由于每个数据成员的访问权限都是 public，因此，对结构成员的访问可以通过对类中对应的数据成员的访问来实现。

2）联合定义的映射

IDL 中的联合的映射处理是比较复杂的。上述 Test 模块中定义的联合 UnionType 被映射为如下形式的 Java 类。

```
public final class UnionType
{
    private short _discriminator;              //区分符
    private java.lang.Object _value;           //存放联合值的对象
    public UnionType()                         //构造函数
    {…}
    public short discriminator()throws org.omg.CORBA.BAD_OPERATION
    {  …
      return _discriminator;                   //获取区分符
    }
    public String a()throws org.omg.CORBA.BAD_OPERATION
    {  …
      switch(_discriminator)
      {
          case 1:
              break
          default:
              throw new org.omg.CORBA.BAD_OPERATION();  //出错
      }
      return((org.omg.CORBA.StringHolder)_value).value;
    }
    public void a(String value)                //为 a 元素赋值
```

```
    {  …
       _discriminator = 1;                    //设置区分值为 1
       _value = new org.omg.CORBA.stringHolder(value);
    }
    ⋮
}
```

从上面的映射结果不难看出，IDL 中的联合被映射为与其同名的 Java 语言的 final 类。同时，在类中为每个数据成员都定义了 get 方法（比如，String a()）和 set 方法（比如，void a (String value)）。在 get 方法中先判断区分符，如果正确的话，返回对应的值，否则抛出异常信息；在 set 方法中，先设置区分符的值，然后再将相应的值保存到数据成员中。尽管在上述映射结果中只给出了 string 类型的数据成员 a 的映射结果，但从中可以看出联合的映射特点有两个：一是利用区分符来标识当前所保存的数据类型，这是由于在任一时刻联合变量中只能保存一种类型的数据；二是所有数据成员都被保存在 java.lang.Object 类型的成员变量中，这是由于在 Java 语言中，java.lang.Object 是所有类的基类的缘故。这里，org.omg.CORBA.BAD_OPERATION 是 CORBA 定义的系统异常，它表示请求了一个不正确的操作。

3）枚举定义的映射

上述 Test 模块中定义的枚举 Day 被映射为如下形式的 Java 类。

```
public final class Day
{
    //与枚举元素对应的各常量的定义
    public static final int _Sunday = 0,
                            _Monday = 1,
                            _Tuesday = 2,
                            ⋮
                            _Saturday = 6;
    //与枚举元素对应的各对象的定义
    public static final Day Sunday = new Day(_Sunday);
    public static final Day Monday = new Day(_Monday);
    ⋮
    public static final Day Saturday = new Day(_Saturday);
    public int value()
    {
        return _value;          //返回枚举元素值(整数)
    }
    //根据给定的整数来获取相应的对象:
    public static final Day from_int(int i)throws org.omg.CORBA.BAD_PARAM
    {
        switch(i)
        {
            case _Sunday:
                    return Sunday;
            case _Monday:
                    return Monday;
            ⋮
            case _Saturday:
                    return Saturday;
            default:
```

```
                throw new org.omg.CORBA.BAD_PARAM();
        }
    }
    private Day(int _value)     //构造函数
    {
        this._value = _value;
    }
    private int _value;         //存放枚举元素值的整型变量
}
```

从上面的映射结果不难看出，IDL 中的枚举被映射为与其同名的 Java 语言的 final 类。这里，org.omg.CORBA.BAD_PARAM 是 CORBA 定义的系统异常，它表示传递了一个不正确的参数。

7. sequence 的映射

sequence 被映射为 Java 语言中的数组。

假如有下述 IDL 定义：

```
sequence<long,20>x;
```

该 IDL 定义的 Java 映射结果如下：

```
int x[20];
```

8. string 和 wstring 的映射

string 和 wstring 都被映射为 Java 语言中的 String 类。

9. 数组的映射

IDL 中的数组被映射为 Java 语言中的数组。

假如有下述 IDL 定义：

```
long LArr[2][3];
```

该 IDL 定义的 Java 映射结果如下：

```
int LArr[2][3];
```

10. 整型量的映射

IDL 中的整型量包括 short 和 long 等。表 3.11 给出了 IDL 整型量的定义及其可能的映射结果。

表 3.11 IDL 整型量的 Java 映射结果

IDL 数据类型	Java 数据类型	IDL 数据类型	Java 数据类型
short	short	unsigned long	int
unsigned short	short	long long	long
long	int	unsigned long long	long

由于 Java 语言中没有无符号整数，所以对无符号数的处理需要在程序中进行。

11. char 和 wchar 的映射

IDL 中的 char 和 wchar 类型都被映射为 Java 中的 char 类型。

12. boolean 类型的映射

IDL 中的 boolean 类型被映射为 Java 语言中的 boolean 类型。

13. octet 类型的映射

IDL 中的 octet 类型被映射为 Java 语言中的 byte 类型。

假如有下述 IDL 定义：

```
octet x;
```

该 IDL 定义的 Java 映射结果如下：

```
byte x;
```

14. any 类型的映射

IDL 中的 any 类型被映射为 org. omg. CORBA. Any 类型。

15. fixed 类型的映射

IDL 中的 fixed 类型被映射为 Java. lang. BigDecimal 类型(但有的系统不可用)。

假如有下述 IDL 定义：

```
typedef fixed<20,2>Dollar;  //整数 18 位,小数 2 位的固定小数类型
Dollar Deposit(in Dollar amount);
```

该 IDL 定义的 Java 映射结果如下：

```
//存款 amount = $250.00
BigDecimal amount = BigDecimal.valueOf(25000,2);
//调用 Deposit
BigDecimal newBalance = account.Deposit(amount) ;
```

16. 浮点类型的映射

IDL 中的浮点类型包括 float 和 double 等。表 3.12 给出了 IDL 浮点类型的定义及其可能的映射结果。

表 3.12　IDL 浮点类型的 Java 映射结果

IDL	Java
float	float
double	double
long double	没有对应类型

3.4.5 IDL 映射后的使用

前面介绍了从 IDL 定义到 Java 语言的映射过程。这里需要注意的是，语言映射的目的是为了在实现客户端或服务器程序时能够使用这些映射结果。下面简单介绍几个 Java 映射结果的使用过程。主要介绍用户定义类型的映射结果的使用，所使用的结构类型定义、联合类型定义和枚举类型定义及其 Java 语言映射等请参考“用户定义类型的映射”部分内容。

1. IDL 结构定义映射结果的使用

下面的程序段给出了利用 IDL 结构定义映射结果的过程。

```
try
{
    StrType c = new StrType(50,"ABC");
    ⋮
    System.out.println(c.field1);
    System.out.println(c.field2);
    ⋮
}
catch(…)
{
    ⋮
}
```

由上述使用过程可知，对 IDL 结构定义映射结果的使用，同使用 Java 语言中的一般的类是一样的，先利用带有参数的构造函数来定义对象，在需要时，可直接利用对象来访问其中的数据成员，这是由于每个数据成员的访问权限都是 public。

2. IDL 联合定义映射结果的使用

对 IDL 联合定义映射结果的使用是由两部分组成的：一是写操作，二是读操作。

```
//写操作
try
{
    UnionType u = new UnionType();
    String  s = "String data";
    u.a(s);                          //设置区分符并保存数据
    ⋮
}
catch(…)
{…}
//读操作
try
{
    switch(u.discriminator())    //判断其中的数据类型
    {
        case 1:
```

```
                String s = u.a();
                System.out.println(s);
                break;
            case 2:
                ⋮
        }
    }
    catch(…)
    {…}
```

对联合变量的使用可分为两部分，一是读操作，另一个是写操作。在读操作之前先要根据区分符来判断联合变量中当前所保存的数据类型，根据该类型来读取对应的数据；在写操作之前先要生成对象，然后调用相应的 set 方法（比如，a(String)）向联合对象中保存数据，需要注意的是，在保存数据之前先要设置区分符。

3. IDL 枚举定义映射结果的使用

对 IDL 枚举定义映射结果的使用也是由两部分组成的：一是写操作，二是读操作。

```
//写操作
try
{
    Day day = Day.from_int(Day._Sunday);  //_Sunday 是静态变量
    ⋮
}
catch(…)
{…}

//读操作
try
{
    ⋮
    String dayString;
    switch(day.value())
{
    case Day._Sunday:
        dayString = "Sunday";
        break;
    case Day._Monday:
        dayString = "Monday";
        break;
    ⋮
    case Day._Saturday:
        dayString = "Saturday";
        break;
    default:
        dayString = "Invalid value";
    }
}
catch(…)
```

{…}

与联合变量的使用类似，对枚举变量的使用也是由两部分组成的：一是写操作，二是读操作。写操作主要就是根据枚举常量来创建枚举对象。读操作主要就是根据枚举对象中的枚举常量所对应的值来获取所对应的枚举常量的名称。

在学过本章的内容之后，就可以继续学习后续章节的内容了，其中包括 CORBA 客户端设计和服务器设计。在这些章节中将具体介绍 CORBA 应用系统的程序设计过程。

第4章 CORBA 客户端程序设计

CORBA 程序设计是由两部分组成的：一是客户端程序设计，另一个是服务器程序设计。就分布式对象系统(包括 CORBA)而言，所谓客户端是指利用所获取的分布式对象引用来调用分布式对象功能的程序。所谓服务器，是指实现分布式对象功能的程序。在 CORBA 系统中，客户端和服务器是分别在不同的进程中运行的。同时，客户端程序和服务器程序可以分别独立地进行开发，而将客户端和服务器联系起来的则是 IDL 接口定义，也就是说，客户端可根据 IDL 定义来调用分布式方法，而服务器则可以根据 IDL 定义来实现分布式对象的功能。

本章主要介绍 CORBA 客户端程序设计的一般方法。为了叙述方便，先引入一个简单的例子，并给出其问题描述和 IDL 定义，该例子将被用于客户端程序设计和服务器程序设计的流程介绍之中。

4.1 问题描述与 IDL 定义

这是利用 CORBA 来实现银行系统的一个假想的简单例子。该系统的客户端部分既可以以 Java 应用程序(Java Application)的形式来实现，也可以以 Java 小程序(Java Applet)的形式来实现，此时 Java 小程序可以被嵌入在 Web 页面上执行。该系统的服务器部分可以以 Java 应用程序的形式来实现。

对于一个简单的 CORBA 银行系统来讲，可以包含如下一些基本功能：

(1) 打开一个账户；

(2) 关闭一个账户；

(3) 存款余额的显示；

(4) 从账户中取款；

(5) 向账户中存款。

利用 IDL 定义上述银行系统的接口如下：

```
module Bank
{
    interface Account;
    interface Control                                    //Control 接口定义
    {
        exception AccountNotExist{ };                    //异常定义
        Account openAccount(in string acct,in string passwd)
```

```
                raises(AccountNotExist);          //根据口令打开账户
        void closeAccount(in string acct);        //关闭一个账户
    };
    interface Account
    {
        readonly attribute unsigned long balance;          //保存余额
        unsigned long Deposit(in unsigned long amount);    //存款并返回余额
        unsigned long Withdraw(in unsigned long amount);   //取款并返回余额
    };
};
```

下面给出 Bank 模块中的 Control 接口的部分映射结果如下：

```
package Bank;
public interface Control extends org.omg.CORBA.Object
{
    public Bank.Account openAccount( java.lang.String acct,java.lang.String passwd)
                        throws Bank.ControlPackage.AccountNotExist;
    public void closeAccount(java.lang.String acct);
    ⋮
}
```

Account 接口中的部分映射结果如下：

```
public interface Account extends org.omg.CORBA.Object
{
    public int balance ();              //只读属性 balance 的映射
    public int Deposit(int amount);
    public int Withdraw(int amount);
    ⋮
}
```

在本章的后续章节中，将在上述 IDL 接口定义及其映射结果的基础上来介绍 CORBA 客户端的实现过程。本章的程序并不完整，只是通过这样一个简单的例子来介绍 CORBA 客户端和服务器的实现过程，第 7 章将完整地介绍几个 CORBA 应用实例的实现及其运行过程。

4.2 CORBA 客户端的组成

如前所述，CORBA 客户端是指通过利用 CORBA 分布式对象的功能来实现特定功能的程序。图 4.1 给出了 CORBA 客户端程序的基本构成。

图 4.1 客户端的处理流程

由图 4.1 可知，CORBA 客户端是由 ORB 的初始化、分布式对象引用的获取以及分布式对象的利用等几部分组成的。

为了加深对上述客户端处理流程的理解，同时也为了便于后面的介绍，先给出 4.1 节中描述的问题的一种客户端程序的实现框架。

```
public class CorbaApplication
{
```

```
    public static void main(String args[ ])
    {
        try
        {
          //ORB 的初始化
          org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
          //分布式引用的获取
          Bank.Control con_ref = Bank.ControlHelper.bind(orb,"BANK");
          // 分布式方法的启动
          Bank.Account acct_ref = con_ref.openAccount(acctno,passwd);
        }
        catch(org.omg.CORBA.SystemException e1)
        {
          //异常处理(标准异常)
        }
        catch(Bank.ControlPackage.AccountNotExist e2)
        {
          //用户定义异常的处理
        }
    }
}
```

这里,是利用 Java 应用程序的形式来实现 CORBA 客户端程序的。同时,在服务器端是以 BANK 为名字来登录分布式对象的。由该程序不难看出,其处理过程与图 4.1 给出的处理流程完全一致。实际上,有的 CORBA 系统的处理过程与这里给出的处理过程还是有区别的,所调用的函数也是不同的,但基本流程是一样的。感兴趣的读者可以利用第 7 章中介绍的 Java IDL 来改写该客户端程序。

下面,将按照客户端的处理流程逐步介绍 CORBA 客户端的程序设计过程。

4.3 ORB 的初始化

在 CORBA 客户端程序中,首先要完成的工作就是 ORB 的初始化。进行 ORB 的初始化主要有如下两个目的:

1. 为进行分布式对象处理而进行的环境初始化

这部分的主要工作就是对分布式对象处理环境进行初始化,只有进行了这一初始化,才能开始分布式对象处理。

2. 为了利用 ORB 接口而获取其对象的引用

为了利用 CORBA 所提供的功能来实现 CORBA 客户端和服务器,可以采用两种方法:一是直接利用 ORB 的引用,另一个是通过利用 ORB 的引用来获取其他接口的引用,进而利用该接口的功能。因此,获取 ORB(接口的)引用是进入 ORB 环境的关键。

在 ORB 中,用于 ORB 初始化的接口定义如下:

```
package org.omg.CORBA;
```

```
public abstract class ORB
{
  :
  public static ORB init();
  public static ORB init(String[ ] args,Properties props);
  public static ORB init(Applet app,Properties props);
}
```

这里的 ORB 接口被定义为抽象类，在该类中与 ORB 初始化有关的方法共有三个，这三个方法都是静态方法，在需要时可以直接通过类名来调用。

第 1 个 init()方法，可用于 Java 应用程序和 JavaApplet，但由于该方法所生成的 ORB 对象，对 JavaApplet 有许多限制，所以，一般在 CORBA 客户端和服务器中利用后面两种方法来生成 ORB 对象。

第 2 个 init()方法用于从 Java 应用程序中来生成 ORB 对象，每调用一次该方法，就生成一个新的 ORB 对象。该方法的第一个参数可用来传递与特定环境有关的参数，第二个参数可设置为标志 ORB 名称的字符串，一般设置为 null。

第 3 个 init()方法，用于从 JavaApplet 中来生成 ORB 对象，每调用一次此方法，就生成一个新的 ORB 对象。该方法的第一个参数一般设为 this，表示当前 applet 对象，第二个参数一般设置为 null。

4.4 ORB 接口的功能

由前述可知，CORBA 客户端程序设计的第一步就是获取 ORB 对象的引用，只有获取了 ORB 对象的引用之后才能够利用由 ORB 接口所提供的方法来实现相应的功能。

ORB 接口一般提供如下几种功能，这些是实现 CORBA 客户端和服务器不可缺少的。

1. ORB 的初始化

如前述那样，ORB 初始化的主要目的就是获取 ORB 对象的引用，以利用其定义的方法来实现相应的功能。

2. 初始服务的初始化

这里的初始服务包括命名服务等由 CORBA 提供的服务。为了利用 ORB 提供的这些服务，首先必须要获取服务对象（如命名服务等）的引用。在 ORB 接口中提供了获取这些服务对象引用的方法。

例如，为了获取命名服务对象的引用，可以利用如下调用语句：

```
orb.resolve_initial_references("NameService");
```

这里，orb 是 ORB 对象的引用。

3. BOA(Basic Object Adapter)的初始化

BOA 是在服务器设计时需要使用的功能，有关 BOA 的基本作用将在后续章节中详细

介绍。BOA 初始化的主要目的就是就是为了获取 BOA 对象引用的。

例如，为了获取 BOA 对象的引用，可以利用如下调用语句：

```
org.omg.CORBA.BOA boa = orb.BOA_init();
```

这里，orb 是 ORB 对象的引用。

4. 分布式对象引用的字符串化处理

所谓分布式对象引用的字符串化处理，就是将分布式对象引用转化为字符串或者将已被字符串化的分布式对象引用恢复为原来的形式。其具体的处理过程将在后续章节中详细介绍。

5. DII 的支持

如前面已经介绍的那样，客户端既可以采用静态的方式来调用分布式方法（利用 Stub 类），也可以采用动态的方式来调用分布式方法（利用 DII）。在 ORB 接口中提供了一组方法来支持客户端的动态调用方式。其具体内容将在后续章节中详细介绍。

6. TypeCode 的支持

为了生成 TypeCode 对象而提供的一组方法。在 ORB 中所处理的数据的类型信息都被保存在 TypeCode 对象中。其具体内容将在后续章节中详细介绍。

7. Java 映射的支持

在 ORB 接口中提供了利用 Java 来进行 CORBA 程序设计所必需的一组方法。这里包括分布式对象的登录/删除以及与 Any 类型有关的处理等。其具体内容将在后续章节中详细介绍。

4.5 分布式对象引用的获取

前面介绍了 ORB 的初始化以及由 ORB 接口所提供的功能。获取 ORB 对象的引用是客户端程序设计的第一步，其目的就是为利用 ORB 的功能做准备。在获取了 ORB 对象的引用之后，为了能够利用分布式对象，就必须要获取分布式对象的引用。实际上，在 CORBA 客户端处理过程中，分布式对象引用的获取是最重要的部分，只有获取了分布式对象引用才能够利用分布式对象的功能。本节主要介绍分布式对象引用的获取方法。

一般来讲，CORBA 系统不同，获取分布式对象引用的方法也可能不同，但到目前为止，大体上可以利用如下几种方法来获取分布式对象引用。

4.5.1 利用文件的方法获取对象引用

利用文件的方法获取分布式对象引用的基本过程是：服务器将所生成的分布式对象引用进行字符串化处理后保存在文件中，并以某种方式将其发送到客户端能够获取的设备上。客户端从该文件中来获取字符串化的分布式对象引用，并将其恢复为原有形式后加以利用。

由该处理过程可知，分布式对象引用的字符串化处理及其逆过程是利用文件的方法来获取分布式对象引用的重要组成部分。

在 ORB 接口中，定义了两个与分布式对象引用的字符串化处理有关的方法：一是 object_to_string()方法，另一个是 string_to_object()方法，这两个方法分别用于将分布式对象引用字符串化和从字符串来生成分布式对象引用。这两个函数的定义形式如下：

```
package org.omg.CORBA;
public abstract class ORB
{
    ⋮
    public abstract org.omg.CORBA.Object string_to_object(String str);
    public abstract String object_to_string (org.omg.CORBA.Object obj);
}
```

利用上述两个方法，以文件为媒体，通过某种通信协议就可以以字符串的形式来获取分布式对象引用。下面介绍其具体用法。

1. 分布式对象引用的字符串化

分布式对象引用的字符串化处理是在服务器端进行的，其具体处理过程如下：

```
//获取 ORB 对象引用
ORB orb = ORB.init();
//将分布式对象引用转化为字符串
String orbstr = orb.object_to_string((Object)con_ref);
//将 orbstr 以文件形式从服务器传递到客户端能获取的设备上
⋮
```

其中，con_ref 是分布式对象引用。

2. 字符串化引用的复原

客户端获取了被字符串化的分布式对象引用之后并不能直接利用，而是需要将其复原后再进行利用，其具体处理过程如下：

```
//从文件中获取字符串化的引用并将其复原
Bank.Control ank_ref;
ank_ref = Bank.ControlHelper.narrow(orb.string_to_object(orbstr));
⋮
```

其中，orbstr 是从服务器传递过来的字符串化的引用。

需要注意的是，object_to_string()方法和 string_to_object()方法所操作的对象都是 org.omg.CORBA.Object 类型的分布式对象引用，因此，在调用 object_to_string()方法时，应该将参数强制转换成 Object 类型。还需要注意的是，在利用 string_to_object()方法复原了分布式对象引用之后，为了能够利用该引用来调用分布式对象中的方法，还必须利用接口 Helper 类中的 narrow()方法，将其转换为可以利用的接口类型，也就是生成对应接口 stub 类的对象（即代理对象）。

利用文件的方式来获取分布式对象引用的优点在于所利用的一些方法，包括 object_to_

string()和 string_to_object()等，都是在根据 CORBA 规范所定义的接口中进行描述的。因此，这种引用获取方法是 CORBA 异种产品之间能够实现分布式对象引用获取的可行的方法之一。

当然，由于在分布式对象引用获取时需要进行文件的读写操作，因此需要依赖于同 CORBA 环境不同的通信手段。

4.5.2 利用 Binding 服务的方法获取对象引用

所谓 binding 服务，实际上是一种提供这样两种功能的服务：一是定位(locating)功能，二是连接(binding)功能。

所谓定位功能，是指查找满足给定条件的分布式对象的功能。连接服务中的定位功能是属于一种命名服务。这里，通过给定能够确定分布式对象的有关信息来查找分布式对象。对客户端而言，需要解决的问题是为了查找一个分布式对象需要提供什么样的检索条件，也就是如何确定一个分布式对象。

所谓连接功能，是指在客户端与服务器的分布式对象之间建立连接的功能。

一般来讲，为了准确确定一个分布式对象，必须要提供如下四种信息：

(1) 分布式对象所实现的接口信息。由于定义分布式对象的目的是为了实现某个接口，因此，接口信息是查找分布式对象的最低条件。

(2) 执行分布式对象的主机地址。对于 TCP/IP 网络来讲，也就是唯一确定一台机器的 IP 地址。

(3) 执行分布式对象的服务器进程的识别信息。也就是识别服务器进程的端口号(Port)。

(4) 服务器进程上的分布式对象的识别信息。也就是能够识别服务器进程上的某个分布式对象的对象键(object key)。

在上述能够确定分布式对象的信息当中，有许多是应用程序无法获取和操作的，比如对象键一般是由 ORB 来分配的，在应用程序中是很难直接对其进行操作的。因此，在应用程序中通过指定上述检索条件来确定一个分布式对象是比较困难的。

鉴于上述原因，在一般的 ORB 产品中都提供了能够简单地识别服务器进程和分布式对象的信息指定方法，这些方法都是由具体的 ORB 产品规定的，一般都是通过字符串的形式来指定的，这些信息被用于分布式对象的定位操作。

这里需要注意的是，利用连接服务来获取分布式对象引用时只需要非常简单的接口。也就是说，客户端利用非常简单的接口就可以获得分布式对象的引用。换句话说，只要执行一个连接方法，就可以获取分布式对象引用。这也是提供连接服务功能的目的。

下面，以一个具体的连接方法为例，介绍连接方法的具体定义形式及其用法。

```
import java.lang.*;
import org.omg.CORBA.*;
public class <接口名>Helper
{
    public static final <接口名> bind(ORB orb){};
    public static final <接口名> bind(ORB orb,String name){};
    public static final <接口名> bind(ORB orb,String name,
```

```
                                    String host,BindOperations options){} ;
}
```

由上面程序段不难看出，连接方法一般是在 IDL 接口定义被映射之后所生成的 Helper 类中定义的，这实际上是默认提供了分布式对象所实现的接口信息，也就是接口仓库 ID (Repository ID)。在上述程序段中，共给出了 3 个 bind 方法定义，在这些方法中除了 ORB 对象引用之外，需要明确给出的定位条件包括 name 参数的对象名以及 host 参数的主机名等。

这里，主机(host)名只要利用域名即可。

对象名(name)需同时指定服务器进程与分布式对象。就上述程序段而言，是利用一个字符串来识别一个分布式对象的(这就是将连接服务归属于一种命名服务的原因)。这里，对象名是在生成分布式对象时，由服务器应用程序设定的字符串，在客户端程序中利用同样的对象名(字符串)来进行连接操作。

由于对象名是由应用程序来设置的，因此，所有的分布式对象都被指定为不同的名字是应用程序设计者的责任，而不是 ORB 的责任。对于小规模的系统来讲，对象名的重复问题不会成为大问题，而对于大规模的系统来讲，对象名的重复问题就会出现了。

客户端应用程序调用 bind()方法时，ORB 就向检索引擎提出查找分布式对象的请求。需提供的检索条件包括必需的 Repository ID、对象名和主机名等。经过定位操作，其结果是获取了分布式对象的引用，以后的处理就是以该分布式对象的引用为基础的。

在获取了分布式对象引用之后就可以进行连接操作，但实际上并不是在此阶段进行连接操作的，通常连接操作是在调用分布式方法时进行的。

为了对连接进行管理以及进行调用请求的传递，在这一处理过程的最后阶段要生成与此接口对应的 Stub 类的对象，即代理对象，也就是根据所获取的分布式对象引用来生成 stub 类的对象。这一处理过程一般是利用 IDL 接口定义经映射之后所生成的 Helper 类中的 narrow()方法来完成的。

4.5.3 利用命名服务的方法获取对象引用

现在的 ORB 产品供应商都提供 CORBA 命名服务。如果从可移植性和互操作性等方面进行考虑，CORBA 命名服务都是最优秀的。

为了利用 CORBA 命名服务，需要利用两个接口：一个接口是为了获取命名服务的对象引用而需要的初始化接口；另一个是命名服务本身的接口。这里需要注意的是：命名服务本身就是作为分布式对象实现的。为了利用命名服务来管理分布式对象引用，首先必须要获取命名服务对象。CORBA 系统不同，命名服务对象的获取方法也可能不同。例如，下面是一种获取命名服务对象的方法：

```
org.omg.CORBA.Object nRoot = orb.resolve_initial_references("NameService");
NamingContext rootNameContext = NamingContextHelper.narrow(nRoot);
```

其中，orb 是 ORB 对象的引用，作为参数的 NameService 是代表命名服务的字符串，rootNameContext 是命名服务对象。

获取了命名服务对象之后，就可以利用命名服务对象所提供的方法来管理和获取分布

式对象引用。

命名服务就如同使用DNS(Domain Naming Service)从机器名来获取IP地址一样,在CORBA中,可以以对象的名字来获取对象引用,它是由CORBA系统所规定的函数来实现的。

4.5.4 利用factory对象的方法获取对象引用

这种方法同前三种方法不太一样,同前三种方法可以并行使用。前三种方法都是为了获取最初的分布式对象引用而被利用的。而利用factory对象的方法,则是通过调用已经获取的分布式对象引用中的方法,来获取新的分布式对象引用。

所谓factory(工厂)对象,是指具有提供新的分布式对象引用的方法的分布式对象。在本章4.1节所描述的问题中,由于在Control接口中定义的openAccount()方法能够返回Account分布式对象的引用,因此,实现Control接口的分布式对象就被称为factory对象。

利用factory对象来获取分布式对象引用的过程是:首先利用上述三种方法中的任意一个来获取factory对象的引用,然后在此基础上,通过调用factory对象中的方法来获取别的分布式对象引用。

通过factory对象来获取分布式对象引用的方法在CORBA应用系统设计中是经常被利用的,它是获取新的分布式对象引用最直接的方法。

4.6 Stub类的构造

在前面的叙述当中,已多次提到"分布式对象引用"这一概念。客户端要想调用分布式对象中的方法,就必须首先获取分布式对象的引用,这同调用本地对象中的方法是一样的。但就CORBA客户端来讲,分布式对象引用实际上是由代理对象来表示的。

在这一节中,将就代理对象的功能以及实现代理对象的Stub类的构造进行详细说明。

4.6.1 代理对象的概念

前已述及,为了减少程序设计的复杂性和提高分布式对象调用的透明性,CORBA客户端可以利用代理对象来调用分布式对象中的方法。

所谓代理对象(proxy object),是指由于不能直接访问存在于别的进程中的分布式对象而设置的分布式对象的代理。在程序中通过代理对象来接收调用请求,并将该请求发送给分布式对象。也就是说,代理对象为程序设计者屏蔽掉了非常复杂的分布式对象调用过程,客户端程序就像调用本地对象一样来调用分布式对象。

在Java环境下,代理对象是作为Java类来实现的,该类一般被称为Stub类。在客户端与分布式对象进行连接操作时,是通过利用所生成的Stub类的对象来进行的。由于这一对象起到代理的作用,故被称为代理对象。

就客户端而言,所谓分布式对象引用,实际上就是这一代理对象的引用。因此,在Java语言的处理上,分布式对象引用就如同Java本地对象引用是一样的。然而,从语义的角度上来讲,分布式对象引用与本地对象引用还是有区别的。

4.6.2 分布式对象引用与本地对象引用的区别

分布式对象引用与一般本地对象引用的最大区别在于对对象引用进行操作时的语义上的差异。

比如，在对两个对象的引用是否相同的比较以及对对象引用进行复制等操作时，对Java本地对象引用来讲，是利用Java.lang.Object类定义的方法来进行的，而对于CORBA对象引用来讲，对上述操作都规定了特殊的意义以及相应的接口，其基本操作方法是在org.omg.CORBA.Object接口中定义的。有关org.omg.CORBA.Object接口中的具体内容将在后续章节中详细介绍。

除了上述差异之外，作为分布式对象引用的代表，Stub类需要完成分布式对象代理的功能，因此在Stub类中需要保存与分布式对象进行通信的信息，同时需要提供实现请求传递的方法。

4.6.3 Stub类的构造

前面多次提到作为代理对象的Stub类，同时也提到Stub类是IDL接口经编译之后产生的。本节通过一个具体实例详细介绍Stub类的构造。

图4.2给出了以Java映射为基础的Stub类的构造。

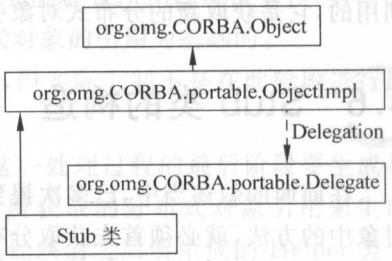

图4.2 Stub类的构造

由图4.2可知，Stub类是通过继承org.omg.CORBA.portable.ObjectImpl类而生成的，同时，在org.omg.CORBA.portable.ObjectImpl类中需要实现在org.omg.CORBA.Object接口中定义的方法，但它并不是直接实现的，而是通过委托(delegation)的方式在org.omg.CORBA.portable.Delegate类中实现的。

在Java语言中，所谓委托(delegation)处理方式，是指不是在本类中实现接口的功能，而是利用其他的类来实现接口的功能。如图4.2所示，由于不是在ObjectImpl类中直接实现org.omg.CORBA.Object接口中定义的方法，而是利用org.omg.CORBA.portable.Delegate类来完成这一实现工作的，因此org.omg.CORBA.portable.Delegate类被称为委托类。

下面，通过给出ObjectImpl类的具体代码来了解委托类的定义方式。

```
public abstract class org.omg.CORBA.portable.ObjectImpl
        extends java.lang.Object implements org.omg.CORBA.Object
{
    //委托(delegate)对象的存储区域和访问方法
    private org.omg.CORBA.portable.Delegate _ _delegate;
    public org.omg.CORBA.portable.Delegate _get_delegate() {}
    public void _set_delegate(org.omg.CORBA.portable.Delegate) {}
    //实现 org.omg.CORBA.Object 接口中定义的方法
    public org.omg.CORBA.ImplementationDef _get_implementation()
    {
```

```
        //调用委托类中的同名方法
        return _get_delegate().get_implementation();
    }
    //其他方法的实现与_get_implementation()方法相同
    ⋮
}
```

由上述 ObjectImpl 类的定义可知，该类用于实现 org. omg. CORBA. Object 接口中定义的方法，但实际的处理过程则是通过委托的方式来完成的。所谓委托(delegation)是指不利用继承的方法而进行功能再利用的一种手段，在 Java 语言程序设计过程中被广泛地应用。

委托的实现过程比较简单，如上述程序所示，把委托类的引用存放在自己的类中，把对自己类的调用请求转换为对委托类的调用请求即可。在上述类的定义中，ObjectImpl 类与其委托类(即 org. omg. CORBA. portable. Delegate 类)具有完全相同的方法，这是定义委托类的前提所在。使用委托类的主要原因是由于 Java 语言只允许单一继承。

了解了 Stub 类的构造之后，下面具体看一下 Stub 类的程序代码结构，并对其实现过程进行简单的分析。所给出的 Stub 类是 4.1 节 IDL 定义中的 Control 接口经编译之后所生成的，由于 CORBA 系统不同，所生成的 Stub 类也不同，这里给出的例子只是 Stub 类的一种可能形式。

```
public class _Portable_Stub_Control extends org.omg.CORBA.portable.ObjectImpl
                        implements Bank.Control
{
    //返回仓库 ID 的方法
    public java.lang.String[ ] _ids()
    {
        return _ _ids;
    }
    //接口的仓库 ID
    private static java.lang.String [ ] _ _ids = {"IDL:Bank/Control:1.0"}
    //openAccount 方法
    public Bank.Account openAccount(String acct,string passwd)
                        throws Bank.ControlPackage.AccountNotExist
    {
      //(1) 生成 Request 对象并设置请求参数
      org.omg.CORBA.Request _request = this._request("openAccount");
      //(2) 向请求对象中设置返回值类型
      _request.set_return_type(Bank.AccountHelper.type());
      //(3) 从 request 对象中申请 Any 变量
      org.omg.CORBA.Any $acct = _request.add_in_arg();
      //向 Any 变量中插入参数 acct
      $acct.insert_string(acct);
      //(4) 从 request 对象中申请 Any 变量
      org.omg.CORBA.Any $passwd = _request.add_in_arg();
      //向 Any 变量中插入参数 Passwd
      $passwd.insert_string(passwd);
      //(5) 向 Request 对象中设置异常类型
      _request.exceptions().add (Bank.ControlPackage.AccountNotExistHelper.type());
```

```
        //(6) 将 request 对象发送给服务器
        _request.invoke();
        //(7) 检查异常信息
        java.lang.Exception _exception = _request.env().exception();
        if(_exception! = null)
        {   //异常处理
            ⋮
        }
        //(8) 取出返回值
        Bank.Account _result;
        _result = Bank.AccountHelper.extract(_request.return_value());
        return _result;
    } //openAccount 方法结束
    //closeAccount()等方法的处理过程与 openAccount()方法类似
    ⋮
}
```

说明：

(1) 以字符串 openAccount 为参数调用在 ObjectImpl 类中定义的_request()方法，来生成 Request 对象(即请求对象)，也就是创建 openAccount()方法的请求对象。

(2) 利用请求对象中的 set_return_type()方法，向请求对象中设置 openAccount()方法的返回值类型。

(3) 利用请求对象中的 add_in_arg()方法来生成 Any 对象，并将调用 openAccount()分布式方法的第一个参数 acct 插入到所生成的 Any 变量中。这里，add_in_arg()方法用于生成处理 in 属性参数的 Any 类型对象，如果需要处理 out 等属性参数的 Any 对象，则需要利用其他方法。

(4) 利用请求对象中的 add_in_arg()方法来生成 Any 对象，并将调用 openAccount()分布式方法的第二个参数 passwd 插入到所生成的 Any 变量中。

(5) 由于 openAccount()方法需要抛出异常，所以需要向请求对象中设置异常类型。

(6) 利用请求对象中的 invoke()方法将请求对象转换为请求信息并进行 Marshaling 操作后将其发送给服务器。需要注意的是，这里的 invoke()方法是需要进行同步调用时所利用的方法，也就是在发出调用 openAccount()方法的请求后，客户端需要处于同步等待状态，直到该方法执行完了为止。如果不需要进行同步调用时，就需要采用其他的方法来发送请求信息。比如，对于 oneway 方法，就需要利用 send_oneway()方法来发送请求信息。

(7) 当 openAccount()方法执行完了之后，就需要检查是否发生了异常，如果发生了异常，就需要对异常进行处理。

(8) 如果没有异常发生，就需要利用请求对象中的 return_value()方法来获取返回值。由于返回值被保存在 Any 对象中，因此需要利用 Account 接口的 Helper 类中的 extract()方法，从 Any 对象中抽取出特定类型的返回值。至此，在 Stub 对象中对 openAccount()方法的一系列处理过程就结束了。

在 Stub 类中，对于 Account 接口中定义的其他方法(如 closeAccount()方法等)的处理过程与 openAccount()方法完全类似。

从 Stub 类的组成可以看出，IDL 接口定义中的每个方法在 Stub 类中都有对应的定义，

但该定义并不是实现对应方法的具体功能，而是将调用请求传递给服务器，由服务器中的对应方法来完成具体的功能。因此，将 Stub 类的对象称为代理对象。

4.7 org.omg.CORBA.Object 接口

在前面的叙述过程中已经多次提到 org.omg.CORBA.Object 接口，它是所有分布式对象都要实现的接口。在 org.omg.CORBA.Object 接口中定义了许多方法，其中有不少方法是用于对分布式对象引用进行操作的，因此又被称为分布式对象引用的语义。

下面，简要介绍 org.omg.CORBA.Object 接口的基本内容。

```
public interface org.omg.CORBA.Object extends java.lang.Object
{
    //(1) 获取接口仓库对象引用
    public abstract org.omg.CORBA.InterfaceDef _get_interface() { }
    //(2) 获取实现仓库对象引用
    public abstract org.omg.CORBA.ImplementationDef _get_implementation() { }
    //(3) 对象引用的复制与释放(管理引用计数值,从而对其生命周期进行跟踪)
    public abstract org.omg.CORBA.Object _duplicate(){ }
    public abstract org.omg.CORBA.Object _release(){ }
    //(4) 对象引用的接口检查(检查当前对象引用是不是某个接口的对象引用)
    public abstract boolean _is_a(java.lang.String id){ } //id 为接口仓库 ID
    //(5) 分布式对象是否存在的检查
    public abstract boolean _non_existent() { }
    //(6) 对象引用同一性检查与 Hash 处理(当前对象与参数所指对象是否相同)
    public abstract boolean _is_equivalent(org.omg.CORBA.Object obj){}
    public abstract int _hash(int max){}//生成识别分布式对象引用所需的值
    //(7) 请求对象的生成
    public abstract org.omg.CORBA.Object _request(java.lang.String op);
    public abstract org.omg.CORBA.Object _create_request(
        org.omg.CORBA.Context ctx,java.lang.String op,
        org.omg.CORBA.NVList nvl,org.omg.CORBA.NamedValue nv);
    public abstract org.omg.CORBA.Object _create_request(
        org.omg.CORBA.Context ctx,java.lang.String op,
        org.omg.CORBA.NVList nvl,org.omg.CORBA.NamedValue nv
        org.omg.CORBA.ExceptionList ex,org.omg.CORBA.ContextList ctc);
}
```

说明：

(1) 获取接口仓库对象引用。

_get_interface()方法用于获取由当前对象引用实现的接口信息。由于该方法的返回值类型 InterfaceDef 是接口仓库(interface repository)的对象引用，所以利用此对象引用就能够访问存储在接口仓库中的信息。

接口仓库主要用于保存 IDL 接口定义的信息，它是在编译 IDL 接口时被保存的。

(2) 获取实现仓库对象引用。

_get_implementation()方法用于获取保存在实现仓库中的信息。由于该方法的返回值类型 ImplementationDef 是实现仓库(implementation repository)的对象引用，因此利用此

对象引用就能够访问存储在实现仓库中的信息。

与接口仓库不同，实现仓库中的信息是需要用户预先登录的。实现仓库中的内容主要包括这样几种信息：一是服务器注册、活动状态、执行及控制等信息；二是对象实例活动状态、执行及控制等信息；三是各种附加信息，包括管理信息和资源分配信息等。

(3) 分布式对象引用的复制与释放。

_duplicate()方法用于对分布式对象引用进行复制操作，_release()方法用于释放分布式对象引用。利用这两个方法就能够对分布式对象引用的引用计数值进行管理，从而能够对分布式对象的生命周期(Lifecycle)进行有效地跟踪。

(4) 分布式对象引用的接口检查。

_is_a()方法用于检查当前对象的引用是不是能够支持由参数指定的仓库 ID 所对应的接口。例如：

```
if( obj._is_a("IDL:Bank/Account:1.0"))
{
    ⋮
}
```

该程序段用于检查 obj 对象是不是实现 Bank 模块中的 Account 接口的对象，如果是，返回 true，否则返回 false。_is_a()方法的参数是仓库 ID。所谓仓库 ID 是指由 IDL 所定义的接口或方法的全局名称。主要用于识别在接口仓库中的接口和操作的，用户可以自己指定，如果不指定的话，接口的仓库 ID 将采用如下形式的字符串来表示：

```
IDL:模块名/接口名:1.0
```

(5) 分布式对象是否存在的检查。

_non_existent()方法用于检查当前分布式对象引用所指向的分布式对象是否存在。该方法在执行过程中必须要访问服务器。

提供_non_existent()方法的原因是由于客户端即使获得了分布式对象引用，但与其对应的分布式对象也可能不存在。这是因为：一般来讲，客户端在获得分布式对象引用的时候，并没有与其对应的分布式对象建立实际的连接，而仅仅是把将来要建立连接时所需要的信息全部保存在代理对象中而已，真正地建立连接是在最初启动方法时进行的，而在获取了分布式对象引用之后到建立实际连接之前这段时间内，分布式对象有可能已经不存在了。

_non_existent()方法主要用于在启动方法之前检查相应的分布式对象是否存在，它相当于一种进行 ping 操作的方法。

(6) 分布式对象引用的同一性检查。

_is_equivalent()方法用于检查作为参数的分布式对象引用与当前分布式对象引用是否都指向同一个分布式对象。

由于语言级(如 Java 语言)的对象引用比较不能用于分布式对象引用的比较，所以系统提供了_is_equivalent()方法来实现此功能。

另外，_hash()方法用于生成识别分布式对象引用所需的数值。与此相同的功能在 Java 语言的 java.lang.Object 类中由 hashcode()方法提供，但这里的_hash()方法对指向同一个分布式对象的分布式对象引用将返回相同的值。

(7) 请求对象的生成。

包括_request()在内的这些方法用于生成在客户端的 DII 动态接口中使用的请求对象(Request Object)。在前面介绍 stub 类时就已经使用过该方法。有关这些函数的用法以及请求对象的具体内容将在后续章节中进行介绍。

4.8 分布式对象方法的启动

客户端处理的最后一步,是通过利用已获得的分布式对象引用来调用分布式对象中的方法。

由于 Stub 类隐藏了分布式处理的实现细节,所以对分布式对象中的方法调用过程,同调用本地 Java 对象中的方法几乎没有什么区别,不同的地方主要有以下两点:

1) 异常处理

第一个需要注意的是异常处理。在 CORBA 中,IDL 可以指定以方法为单位的用户定义的异常处理,同时,CORBA 还规定了系统异常。因此,客户端在调用分布式对象中的方法时,即使该方法没有处理用户定义的异常,也必须要检查 CORBA 的系统异常。就 Java 语言来讲,对 CORBA 异常的处理,也是利用 try-catch 语句来实现的。

2) Holder 类的利用

另一个需要注意的是对 IDL 参数属性的处理。在前面已经提到过,Java 的参数传递方式同 CORBA 的参数传递方式有很大的不同,其中最主要的原因是在 CORBA IDL 的参数传递方式中,需要处理具有 out 和 inout 属性的参数。为了支持这两种参数传递方式,在 Java 映射中提供了 Holder 类,由 Holder 类来实现具有这种属性的参数传递。

下面以一个简单的例子来具体介绍利用 Holder 类进行参数处理的过程。

假如有如下 IDL 接口定义:

```
module test
{
    interface Istruct
    {
        struct sData
        {
            string str;
            long len;
        };
        void getValue( out sData value);
        void setValue( in sData value);
    };
}
```

在上述 IDL 定义中,给出了用户自定义类型 sData 的定义,同时还给出了两个操作(方法)的定义:getValue()和 setValue()。这里需要注意的是,这两个操作的参数都是 sData 结构类型,其中一个具有 in 属性,另一个具有 out 属性。

在客户端程序中要想调用上述接口定义中的 getValue()方法和 setValue()方法,就必须要利用用户定义的结构类型 sData,而要想利用 sData 结构类型,就必须要掌握用户定义

的结构类型的映射方法，有关 IDL 结构类型的映射方法请参见前面章节的内容。下面具体介绍客户端程序对 getValue()方法和 setValue()方法调用过程。

```
//(1) 创建 sData 结构变量，并为其设置初值
test.IstructPackage.sData sDataRef = new test.IstructPackage.sData("Nothing",100);
//(2) 根据所获取的分布式对象引用，调用 setValue()方法
IsRef.setValue(sDataRef);
//进行其他操作
⋮
//(3) 创建 sData 结构类型的 Holder 类
test.IstructPackage.sDataHolder sHolderRef = new test.IstructPackage.sDataHolder();
//(4) 根据所获取的分布式对象引用，调用 getValue()方法
IsRef.getValue(sHolderRef);
//(5) 从 sHolderRef 对象中获取 sData 变量
test.IstructPackage.sData sDataRet = sHolderRef.value;
//(6) 利用 sDataRet.str 和 sDataRet.len 就可以取出结构变量中的对应数据成员
java.lang.String s = sDataRet.str;
int x = sDataRet.len;
⋮
```

说明：理解本程序段的关键是要掌握用户定义类型的映射以及用户定义类型所对应的 Holder 类的定义。由于 CORBA 系统不同，其 IDL 映射结果也不同，因此上述程序段只是一种可能的形式。下面按顺序进行说明。

(1) 为了调用 setValue()分布式方法，需要创建 sData 结构变量，以便作为参数传递给 setValue()方法。回忆一下用户定义的结构类型的映射结果可知，结构类型被映射为同名的 Java 语言中的类，其中定义了两个构造函数：一个是无参的默认构造函数，另一个是具有与结构类型中的数据成员个数相同参数的构造函数。这里是利用带参数的构造函数来创建对象的，以便为对象赋初值。需要注意的是，由于 sData 结构类型是在 Istruct 接口中定义的，而不是直接定义在 test 模块下，因此，在定义该结构类型的变量时，应该注意表示该类型的路径的写法。

(2) 获取了分布式对象引用之后，对分布式对象中的方法调用就如同调用本地对象中的方法一样。

(3) 为了创建 Holder 类对象，需要了解 Holder 类的构造，这同创建一般类的对象是一样的。如前所述，对于结构这样的用户定义类型，其 Holder 类是在进行 IDL 映射时被生成的，而其他一般类型(比如 string 类型)的 Holder 类则是在 CORBA 系统中预先定义好的。使用 Holder 类的目的是为了处理 out 和 inout 属性的参数，以便能够从服务器向客户端传递参数。

(4) 由于 out 属性和 inout 属性的参数被映射为 Java 语言的 Holder 类，因此，在调用具有这种参数属性的方法时需要以 Holder 类对象作为参数。

(5) 由于 Holder 类参数能够从服务器向客户端传递数据，这样在客户端程序中就需要从 Holder 类参数中取出返回数据。如何从 Holder 类中取出数据，完全取决于 Holder 类的结构。对于用户定义的结构类型参数来讲，从 Holder 类取出的应该是这种类型的结构变量。

(6) 根据所返回的结构变量，从中取出对应的数据成员。由于用户定义的结构类型被

映射为 Java 语言的类，因此，对结构成员的访问同对象成员的访问是完全一样的。

4.9 Java Applet 中的 CORBA 客户端结构

前面主要介绍了 Java Application 程序的 CORBA 客户端的实现方法。本节将简单介绍在 Java Applet 程序中 CORBA 客户端的实现过程。

对于 CORBA API 来讲，实现 Java Application 和 Java Applet 基本上没有什么变化，只有一点是必须改变的，这就是在 ORB 初始化时所调用的 init()方法，这一点在介绍 ORB 初始化时已经提到过。其他需要修改的地方只是受 Java Applet 的程序格式的限制。

需要注意的是，在利用 Applet 来实现 CORBA 客户端时，需要考虑安全性问题，这是由其应用环境决定的。

4.9.1 Java Applet 中的 CORBA 客户端结构

根据 Java Applet 的结构要求，其 CORBA 客户端程序一般可采用下面的结构来实现。

```
public class CorbaAppletClient extends JApplet
{
    //数据成员的定义
    ORB orb = null;
    ⋮
    public void init()
    {     //对相应的 GUI 组件进行初始化
          ⋮
         initControl();  //与 CORBA 有关的初始化
    }
    public void initControl()
    {
          //ORB 的初始化
          //分布式对象引用的获取
          //各个 Button 所对应的 ActionListener 的登录
          ⋮
    }
    //定义一些在各 Button 的 ActionListener 处理过程中需要调用的函数
    ⋮
    class OpenActionListener implements ActionListener
    {
        public void actionperformed(ActionEvent e)
        {
              //单击 Open 按钮时的处理过程
              ⋮
        }
    }
}
```

说明：在利用 Applet 来实现 CORBA 客户端时，主要需要按照 Applet 的结构来进行处理。其中包括初始化和 ActionListener 的定义。初始化部分包括 ORB 的初始化和分布式

对象引用的获取等。

4.9.2 ORB 的初始化

如前所述，在 ORB 类中有三个 init()函数，这些函数都定义为静态的，在需要时直接通过类名就可调用。在 Applet 程序中，要调用 init(Applet app,properties props)；函数来进行初始化。该函数的一般调用形式如下：

```
orb = ORB.init(this,null);
```

其中，第一个参数是将 Applet 对象自身传递给它，第二个参数是将在初始化函数中要使用的属性列表传给它，但一般只传递 null 即可，其返回值是 ORB 对象的引用。

4.9.3 分布式对象引用的获取

在介绍 Java 应用程序时，是利用命名服务来获取分布式对象引用的。尽管在 Applet 程序中也可以利用命名服务来获取分布式对象引用，但由于如下两个原因，一般还是采用通过文件的方式来获取分布式对象引用。

(1) 由于执行 Applet 的客户端上安装的 ORB 产品不一定和服务器方的 ORB 产品相同，假如客户端使用的是 VisiBroker 产品，而服务器是采用 OrbixWeb 产品，这样在 Applet 程序中就需利用 VisiBroker 的 ORB 类中的 resolve_initial_references()方法来获取 OrbixWeb 的命名服务对象，这样，Applet 程序的实现就同 VisiBroker 系统有关了。由于 Applet 程序应该能在任何客户端上运行，因此就出现了问题。

(2) 由于考虑到安全性，通常的 Applet 程序(不包括数字签名的 Applet)只能和其所在的 Web 服务器进行连接，这样，要想在 Applet 程序中使用命名服务，命名服务器就必须和 Web 服务器在同一台机器上。

鉴于以上原因，一般在 Applet 程序中不采用命名服务的方式来获取分布式对象引用，这里，介绍利用文件的方式来获取分布式对象引用的方法。为简单起见，只给出在 Applet 程序中的处理过程，在服务器方的处理过程以后再介绍。

正如前面所介绍的那样，采用文件的方式来获取分布式对象的引用时，首先需要在服务器方把相应的分布式对象引用转换为字符串存入文件中并传递给客户端，然后在客户端一方从该文件中读入该字符串，并把它转换为分布式对象引用。

下面的例子给出了从 Web 服务器中读取分布式对象引用所存放的文件的过程。这里假设分布式对象引用是以字符串的形式被存放在 Web 服务器中的 controlRef.dat 文件中的。

```
//从 controlRef.dat 文件中读入 Control 分布式对象引用
Control controlRef = null;
String controlRefString = null;
URL cBase = null;
try
{
    //(1) 获取 Web 服务基地址
    cBase = getCodeBase();
```

```
        //(2) 调用 URL 构造函数来生成与 ControlRef.dat 文件对应的 URL 对象
        URL controlRefFile = new URL(cBase,"ControlRef.dat");
        //(3) 生成 DataInputStream 对象
        DataInputStream istream = new DataInputStream(controlRefFile.openStream());
        //(4) 读入 UTF-8 形式字符串
        controlRefString = istream.readUTF();
        //(5) 关闭 DataInputStream 对象
        istream.close();
    }
    catch(IOException excep)
    {
        system.err.println("File reading Error!");
        system.exit();
    }
    //(6) 将字符串的对象引用转换为 org.omg.CORBA.Object 类型的对象引用
    org.omg.CORBA.Object controlRefObj = orb.string_to_object(controlRefString);
    //(7) 将 org.omg.CORBA.Object 类型的对象引用转换为 Control 类型的对象引用
    controlRef = ControlHelper.narrow(controlRefObj);
    ⋮
```

说明：

(1) 由于 controlRef.dat 文件和 Applet 的类文件是存放在 Web 服务器的同一个目录下的，所以，首先利用 getCodeBase()方法来获取存放 Applet 的目录的 URL 对象(即利用 URL 对象来表示目录)。

(2) 以上面获取的 URL 对象为第一参数，controlRef.dat 文件名为第二参数来调用 URL 类的构造函数，以生成与 controlRef.dat 文件相对应的 URL 对象。

(3) 调用由(2)所生成的 URL 对象的 openStream()方法，以便生成与 controlRef.dat 文件所对应的 InputStream 对象，并以此为基础来生成 DataInputStream 对象(以便从 InputStream 中读入字符串)。

(4) 调用 DataInputStream 对象中的 readUTF()函数，读入 UTF-8 形式的字符串(UTF-8 是 Universal Transformation Format 8 的缩写)。到此为止，读入文件过程就结束了。

(5) 关闭 DataInputStream 对象。

(6) 利用 ORB 的 string_to_object()方法，将字符串形式的分布式对象引用转换为 org.omg.CORBA.Object 类型的引用。

(7) 利用 ControlHelper 类中的 narrow()方法将上述对象引用转换为 Control 类型的引用，也就是生成 Stub 类代理对象，以便可以调用其中的分布式方法。

由于利用 ORB 对象的 object_to_string()方法所生成的字符串类型的对象引用，在 CORBA 规范中对其进行了具体说明，这样，即使客户端与服务器使用不同的 ORB 产品也不会出现问题。

4.9.4 在 HTML 文件中使用 Applet

Applet 程序设计完成之后，就需要在 HTML 文件中来使用 Applet。在 HTML 中使

用 Applet 的大致结构如下：

```
<HTML>
<HEAD>
<TITLE> CORBA Applet Program </TITLE>
</HEAD>
<BODY>
<Applet CODE = "CorbaAppletClient.class" ARCHIVE = "CC.jar " WidTH = 495 HEIGHT = 353></Applet>
</BODY>
</HTML>
```

其中 CC.jar 文件是把所需要的所有文件都打包在一起，是利用 Java 中的现成工具来生成的。例如：

```
jar cf c:\corba\CC.jar
```

其意义是把当前目录下的所有目录和文件都打包到 C:\corba\CC.jar 文件中。

第5章 CORBA服务器程序设计

在分布式对象系统中，所谓服务器是指实现分布式对象功能的程序，它是由两部分组成的：一是分布式对象实现部分，这部分是CORBA服务器的核心部分，用于实现由IDL定义的接口功能；二是服务器进程部分，这部分主要用于实现分布式对象的生成与登录，以及接收来自客户端的调用请求等。CORBA服务器是根据IDL接口定义来实现的。在CORBA服务器的实现过程中，需要利用IDL的Java语言映射结果。本章主要介绍CORBA服务器的实现过程，所利用的IDL接口及其映射结果与4.1节给出的例子相同。

5.1 CORBA服务器的构造

如前所述，CORBA服务器是指实现分布式对象功能的程序。图5.1给出了CORBA服务器程序的基本构成。

由图5.1可知，CORBA服务器部分是由ORB的初始化、BOA/POA的初始化、分布式对象的生成、分布式对象的登录以及接收请求开始等几部分组成的。

为了加深对上述服务器处理流程的理解，同时也为了便于后面的介绍，在这里先给出4.1节描述的一种服务器程序的实现框架。

图5.1 CORBA服务器的处理流程

```
public class ControlSerImpl
{
  public static void main(String args[ ])
  {
      //args 的检查与处理
      try
      {
            //ORB 的初始化
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            //BOA 的初始化
             org.omg.CORBA.BOA boa = orb.BOA_init();
            //分布式对象的生成
            ControlImpl control = new ControlImpl("BANK");
            //分布式对象的登录（登录到 BOA 中）
            boa.obj_is_ready(control);
            //接受请求
```

```
            boa.impl_is_ready();//循环等待
        }
        catch(org.omg.CORBA.SystemException system_exception)
        {
            //异常处理
        }
    }//main end
}
```

这里是利用 Java 应用程序的形式来实现基于 BOA 的 CORBA 服务器程序。同时，在该程序中是以 BANK 为名字来登录分布式对象的。由该程序不难看出，其处理过程与图 5.1 给出的处理流程完全一致。实际上，有的 CORBA 系统的处理过程与这里给出的处理过程还是有区别的，所调用的函数也是不同的，但基本流程是一样的。感兴趣的读者可以利用第 7 章中介绍的 Java IDL 来改写该服务器程序。

下面，将按照服务器的处理流程逐步介绍 CORBA 服务器端的程序设计过程。由于 ORB 的初始化在第 4 章中已进行了说明，故在此不再进行详细叙述。其余部分将按顺序进行讲解。同时先介绍基于 BOA 的程序设计，然后再介绍基于 POA 的程序设计。

5.2 对象适配器的作用

5.2.1 对象适配器的作用

在第 4 章中已经提到过，ORB 的初始化结束以后，就已经获得了 ORB 对象的引用，这样，就可以使用 ORB 的功能了。

在 CORBA 中，把 ORB 功能中的服务器所需要的功能都组织到被称为对象适配器 OA (Object Adapter)的对象中来提供。这也就是说 OA 是 ORB 的一个组成部分，同时，由 OA 所提供的功能也仅在开发服务器程序时使用，而不用于开发客户端程序。

由 CORBA 所提供的服务器的功能主要有如下几种，这些功能在不同的系统中所体现的方式可能是不同的，有的是分别提供的，有的是一个功能中包含多项：

① 分布式对象引用的生成；
② 分布式对象的登录与删除；
③ 服务器进程的登录与删除；
④ 分布式对象的激活；
⑤ 服务器进程的激活；
⑥ 方法调用请求的发送(dispatch)；
⑦ 方法启动请求认证信息的取出。

CORBA 将其服务器的功能独立存放于 OA 之中的原因就在于 CORBA 服务器有各种各样的类型，这些服务器类型所需要的 ORB 的功能也不相同，而且，即使是相同的功能也可能需要不同的接口。尽管满足所有这些要求的接口都可以在 ORB 接口中进行定义，但是更有效的方法是根据服务器的类型的不同，通过各种类型的对象适配器(OA)来提供所需要功能的接口，这样也更易于使用。

5.2.2 BOA 与 POA

在 CORBA 版本 1.0 到版本 2.1 中规定，ORB 产品必须提供的对象适配器是 BOA (Basic Object Adapter，基本对象适配器)。正像其名字所示的那样，BOA 提供了由对象适配器必须提供的最基本的功能。由 BOA 所提供的具体功能，将在下一节中进行介绍。在 CORBA 版本 2.2 中将 BOA 删掉了，取而代之的是 POA(Portable Object Adapter，可移植的对象适配器)。

由 POA 来取代 BOA 主要是基于以下的原因：

(1) 在 CORBA 规范中，对 BOA 的描述比较模糊，不同的开发者可能会产生不同的解释，这样，各种 ORB 产品对 BOA 的实现都有很大的不同。由于在接口定义与使用方法上都不尽相同，其中包括 API 本身的差异，从而难以实现服务器程序的可移植性(portability)。

(2) 由 BOA 所提供的功能难以满足对 CORBA 系统的开发。为此，需要对 BOA 的功能进行扩充。

POA 就是为了解决上述问题而开发的，它是对象适配器的新的标准接口。在 POA 中，所有的接口都是由 IDL 进行严格定义的，以便尽可能排除在接口描述上的模糊性。

在本章的后续内容中将介绍 BOA 和 POA 的功能。

5.2.3 伪对象

在 CORBA 系统中定义了许多伪对象，其中 BOA 本身就是被作为伪对象实现的。从应用程序的角度来看，伪对象与普通的 CORBA 对象没有什么区别，但实际上伪对象并不是作为 CORBA 的分布式对象来实现的，之所以称其为伪对象是由于它的接口的确是利用 IDL 进行定义的，但却是作为一般对象来实现的，比如在支持 Java 映射的产品中，伪对象就是作为 Java 对象来实现的。换句话说，像 BOA 这样的伪对象，并不是通过继承 org.omg.CORBA.Object 接口实现的，而 CORBA 分布式对象的主要特征就在于它是继承 org.omg.CORBA.Object 接口实现的。

概括地讲，伪对象与普通的 CORBA 分布式对象的主要区别有如下几点：

(1) 由于伪对象不是分布式对象，因此不能将伪对象作为一般 CORBA 对象中操作(方法)的参数进行传递。

(2) 在接口仓库中没有伪对象的定义信息。

5.3 BOA 的功能

由于 BOA 本身也是作为对象来实现的，因此，同 ORB 一样，为了利用其功能也必须要先获取其对象引用。为了获取 BOA 对象的引用，一般需要首先获取 ORB 对象的引用，然后通过调用 ORB 对象中的方法来获取 BOA 对象的引用。

在获取了 BOA 的对象引用之后，就可以利用 BOA 的功能来实现服务器的功能了。由于不同的 CORBA 产品对 BOA 的定义也是有差异的，因此，下面给出的只是 BOA 的一种定义形式，从中可以了解由 BOA 所提供的基本功能。

```
public abstract class org.omg.CORBA.BOA extends java.lang.Object
{
    //取出方法启动请求的认证信息
    public abstract org.omg.CORBA.principal
                    get_principal(org.omg.CORBA.Object obj);
    //分布式对象的登录
    public abstract void obj_is_ready(org.omg.CORBA.Object obj);
    public abstract void obj_is_ready(org.ong.CORBA.Object obj,…);
    //分布式对象的删除
    public abstract void deactivate_obj(org.omg.CORBA.Object obj);
    //服务器进程的登录
    public abstract void impl_is_ready();
    public abstract void impl_is_ready(…);
    ⋮
}
```

说明：根据前述的 BOA 的基本功能，来说明 BOA 类的基本内容。

(1) 分布式对象引用的生成。

尽管在上述 BOA 类中并没有给出用于生成分布式对象的引用的函数定义，但在 CORBA 规范的 IDL 定义中给出了 create()和 get_id()等方法，这些方法主要用于生成分布式对象引用等操作。实际上，在程序中一般是不直接利用这些方法的。在现有的 BOA 环境下，基本上都是在分布式对象实现(类)的构造函数中来生成分布式对象引用的。

(2) 分布式对象的登录与删除。

为了使分布式对象能够通过 BOA 接收到来自客户端的“启动方法的请求”，就必须将分布式对象登录到 BOA 中，obj_is_ready()方法就是用于完成这一处理过程的。obj_is_ready()方法的作用只是将分布式对象登录到 BOA 中，而为了使服务器进程能够开始接收调用请求还必须调用 impl_is_ready()方法。

为了删除登录在 BOA 中的分布式对象，以便不接收“启动方法的请求”，就需要使用 deactivate_obj()方法。

(3) 服务器进程的登录。

为了将“服务器进程初始化处理完成”以及可以接收“启动方法的请求”的状态通知 BOA，就需要使用 impl_is_ready()方法。这一方法通常进入等待请求的循环之中，而不返回。在有的 CORBA 系统中还提供了删除服务器进程的方法，比如 deactivate_impl()方法等，该方法的作用就是结束服务器进程对调用请求的接收过程。

(4) 分布式对象的激活。

所谓分布式对象的激活，是指生成分布式对象并使其进入可执行状态。通常服务器进程在生成分布式对象以后就将其登录到 BOA 中，这样，就不需要再单独实施激活操作。实际上，在 CORBA 规范中规定了当服务器进程上不存在分布式对象时的激活功能，但这种功能是因系统而不同的。

(5) 服务器进程的激活。

在 CORBA 中，与激活分布式对象相对应的是提供激活服务器进程的方法。这些方法的实现也是随着产品的不同而不同。一般来讲，在服务器进程登录之后，服务器进程也就激活了。

(6) 方法启动请求的发送。

接收启动方法的请求并将其发送给(dispatch)相应的分布式对象是BOA的功能之一。在此以后,根据分布式对象实现中的skeleton类或者DSI类,最终将调用请求发送给相应的分布式方法。BOA的这一功能,从应用程序的角度是看不到的。

(7) 方法启动请求的认证信息的取出。

对调用请求的发送者是否合法的认证工作一般来讲是由分布式对象实现来完成的。为了进行认证操作,ORB将相当于"请求发行者"身份证明的Principal对象连同"启动方法的请求"一起进行传递,get_principal()方法就是用于取出与请求信息一起传递过来的principal对象。

5.4 分布式对象实现

所谓分布式对象实现是指实现分布式对象功能的代码,对Java映射来讲,一般是用一个Java类来实现的。

在分布式对象实现中必须要实现的功能主要包括如下几个:

(1) 由IDL接口所定义的方法。这是分布式对象实现中需要实现的主要功能,这也是定义分布式对象实现的原因。

(2) 由org.omg.CORBA.Object接口所定义的分布式对象的语义。如同java.lang.Object是所有Java语言本地对象的父类一样,org.omg.CORBA.Object是所有CORBA对象都必须要实现的接口。只是在实际的程序设计过程中并不需要我们直接去实现这些方法,而是可以借用其他的类来实现其功能,这一点将在后续内容中进行介绍。

(3) 保存分布式对象的ID以及连接(connection)管理等信息的方法。

(4) 将来自客户端的请求发送给相应的分布式方法的功能。

前三个功能与代理对象的功能非常相似,但后一个功能也是必须要实现的。

在对本地对象中的方法进行调用时(即调用同一个进程中的对象中的方法),可通过直接利用方法的地址来完成。但是,对于CORBA这样的分布式对象环境来讲,为了调用分布式对象中的方法,需要在代理对象的内部将调用请求转换成对CORBA服务器的请求,由CORBA服务器来接收调用请求,将其发送给相应的分布式对象实现。这一处理过程,是由ORB,更具体地说是由OA来实现的。

如前所述,分布式对象的实现方法有两种:一种是利用Skeleton类,即静态实现方法,另一种是利用DSI,即动态实现方法。本章主要介绍利用Skeleton类的实现方法。

在CORBA中,Skeleton类是由IDL接口定义经编译之后自动生成的,不需要自己来编写。但是,所生成的Skeleton类与自己编写的分布式对象实现必须要按照某种方式结合起来。在Java映射中,这种结合方式有两种:一种是采用Skeleton类继承方式,另一种是采用Tie机制方式。下面将分别进行介绍,并讨论每种实现方式的特点。

5.4.1 Skeleton继承方式

所谓Skeleton继承方式,就是通过直接继承Skeleton类来实现分布式对象实现。这种方

式的特点是由对象适配器(OA)来调用 Skeleton 类中的 dispatch 方法，而 dispatch 方法通过参数来接收来自客户端的调用请求。图 5.2 给出了 Skeleton 继承方式的系统构造示意图。

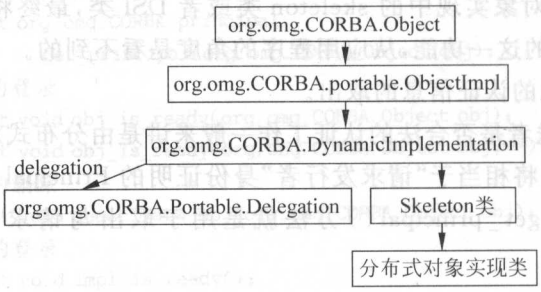

图 5.2 Skeleton 继承方式的构成示意图

下面的例子说明了通过继承 Skeleton 类来实现分布式对象的具体过程。

```
public class ControlImpl extends Bank._ControlImplBase
{
    ControlImpl ( String bank )
    {
        super(bank);
    }
    public Bank.Account openAccount ( String acctid,String passwd)
                    throws Bank.Controlpackage.AccountNotExist
    {
        //openAccount()操作的处理过程
    }
    public void closeAccount( String acctid)
    {
        //closeAccount()操作的处理过程
    }
    ⋮
}
```

说明：

(1) _ControlImplBase 是 Control 接口经过编译之后所生成的 Skeleton 类。

(2) 由上述程序可以看出，通过继承 Skeleton 类来实现分布式对象是非常简单的，对分布式对象的调用细节都被隐藏在 Skeleton 类中，从程序设计的角度来讲，分布式对象的定义过程同本地对象的定义过程基本上是一样的。

5.4.2 Skeleton 类的构造

在 CORBA 服务器程序设计过程中，由于 Skeleton 类的作用使得很多复杂的细节都被隐藏掉了，所以程序设计人员可以集中精力来研究业务逻辑的实现问题，从表面上看，就如同实现本地对象一样来实现分布式对象。

在上一章介绍了 Stub 类的结构，在本节将介绍 Skeleton 类的构成。由于 CORBA 系统不同，所生成的 Skeleton 类也可能是有差异的，下面给出的是在 4.1 节定义的 Control 接口

的一种 Skeleton 类的映射结果。

```
abstract public class _ControlImplBase extends org.omg.CORBA.DynamicImplementation
                                                    implements Bank.Control
{
    //(1) 方法表(利用 hash 表来实现)
    private static java.util.Dictionary _methods = new java.util.Hashtable();
    static //static block:用于对一组 static 变量进行初始化
    {
        _methods.put("openAccount",new java.lang.Integer(0));
        _methods.put("closeAccount",new java.lang.Integer(1));
    }
    //(2) dispatch 方法
    public void invoke(org.omg.CORBA.ServerRequest _request)
    {
        //(3) 取出与方法名对应的索引值
        Bank.Control _self = this;
        java.lang.Object _method = _methods.get(_request.op_name());
        if(_method == null)
        {
            throw new org.omg.CORBA.BAD_OPERATION(_request.op_name());
        }
        int _method_id = ((java.lang.Integer)_method).intValue();
        //(4) 根据方法索引值进行分别处理
        switch(_method_id)
        {
            //(5) openAccount()方法的处理过程
            case 0:
            {
                try
                {
                    //(6) 取出 openAccount()方法所需要的参数
                    org.omg.CORBA.NVList _params = _orb.create_list(0);
                    org.omg.CORBA.Any $acct = _orb.create_any();
                    $acct.type(_orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_string));
                    _params.add_value("acct",$acct,org.omg.CORBA.ARG_IN.value);
                    org.omg.CORBA.Any $passwd = _orb.create_any();
                    passwd.type(_orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_string));
                    _params.add_value("passwd",$passwd,org.omg.CORBA.ARG_IN.value);
                    _request.params(_params);
                    java.lang.String acct;              //取出 account 参数
                    acct = $acct.extract_string();
                    java.lang.String passwd;            //取出 password 参数
                    passwd = $passwd.extract_string();
                    //(7) 调用实际的 openAccount()方法
                    Bank.Account _result = _self.openAccount(acct,passwd);
                    //(8) 设置返回值
                    org.omg.CORBA.Any _resultAny = _orb.create_any();
                    Bank.AccountHelper.insert(_resultAny,_result);
                    _request.result(_resultAny);        //通过_request 对象返回给客户端
                }
```

```
                    //(9) 异常处理
                    catch(Bank.ControlPackage.AccountNotExist _exception)
                    {
                        org.omg.CORBA.Any _exceptionAny = _orb.create_any();
                            Bank.ControlPackage. AccountNotExistHelper.insert(
                                                        _exceptionAny,_exception);
                            _request.except(_exceptionAny);
                        }
                        return;
                    }
                    //(10) closeAccount()方法的处理过程
                    case 1:
                    {
                        //与 case 0 的处理相同,省略
                    }
                } //switch 结束
            }
            //省略
}
```

说明：在此 Skeleton 类_ControlImplBase 中，主要是用于实现 dispatch 功能的。所谓 dispatch，是指根据所传递的参数转去执行相应的功能的过程。就 Skeleton 类来讲，是指根据来自客户端的调用请求，转去执行相应的分布式方法。dispatch 功能的实现，依赖于在程序中定义的方法表(Method table)和 dispatch 方法。下面，根据程序中所标注的序号逐一进行介绍。

(1) 创建方法表。方法表是由方法名及其索引值构成的 hash 表。就本例来讲，由于只有两个方法，即使不利用 hash 表也是可以实现的，但为了支持对规模较大的接口(方法较多)的处理，在 Skeleton 类中，利用 hash 表来保存方法名及其索引值。索引值是按照每个方法在 IDL 定义中的顺序设置的，主要用于后面 case 语句的判断中。需要注意的是，对方法表的初始化是利用 static 块来完成的。

(2) dispatch 方法的定义。这里的 dispatch 方法就是名为 invoke 的方法。此方法的原型是在 DynamicImplementation 接口中定义的，而 DynamicImplementation 则是 DSI 的一部分。下面的程序段给出了 DynamicImplementation 类的定义。

```
public abstract class org.omg.CORBA.DynamicImplementation extends
                                    org.omg.CORBA.portable.ObjectImpl
{
    //dispatch 方法
    public abstract void invoke(org.omg.CORBA.ServerRequest request);
    ⋮
}
```

由 DynamicImplementation 类的定义可知，该类是通过继承 ObjectImpl 类实现的。ObjectImpl 类的内部构造，已经在 CORBA 客户端程序设计一章中详细说明过，它是通过利用 Delegation 类来实现的。在此不再详述。

在 DynamicImplementation 类中，作为 dispatch 方法的 invoke()是抽象方法，必须要在 skeleton 类中对其进行实现。invoke()方法的参数是 ServerRequest 类型的。

下面的程序段给出了 ServerRequest 类的定义。

```
public abstract class org.omg.CORBA.ServerRequest extends java.lang.Object
{
    //取出方法名
    public abstract java.lang.String op_name();
    //取出 context 信息
    public abstract org.omg.CORBA.Context ctx();
    //将来自客户端的调用参数存入入口参数的 NVList 表中
    public abstract void params(org.omg. CORBA.NVList);
    //设置返回结果
    public abstract void result(org.omg.CORBA.Any);
    //设置异常信息
    public abstract void except(org.omg.CORBA.Any);
    //构造函数
    public org.omg.CORBA.ServerRequest();处理
}
```

由 ServerRequest 类的定义不难看出，在 ServerRequest 类中提供了访问来自客户端的调用请求信息以及设置返回结果信息的方法。实际的处理过程则是利用在后续章节中介绍的 Any 对象和 DII 接口等完成的。

(3) 取出与方法名对应的索引值。为了获取调用请求中的方法名，需要利用 ServerRequest 类中的 op_name()方法。

(4) 根据方法索引值进行分别处理。这种处理方式正是 dispatch 的含义。

(5) openAccount()方法的处理过程。这里需要注意的是 openAccount()方法所对应的索引值为 0。

(6) 取出 openAccount()方法所需要的参数。为了从调用请求中取出参数，首先需要生成 NVList 对象_params，同时将所生成的用于接受参数的两个 Any 变量追加到_params 对象中。并以_params 对象为参数来调用 ServerRequest 对象中的 params()，以便将参数表传递给 ServerRequest 对象，从而将来自客户端的调用参数存入 NVList 对象中。在此基础上，就可以从对应的 Any 变量中获取参数值。

(7) 调用实际的 openAccount()方法。在获取了调用参数之后，就可以调用实际的分布式方法了。这里需要注意的是，利用 this 对象所调用的 openAccount()方法是通过继承该 Skeleton 类而实现的分布式对象实现中的方法。

(8) 设置返回值。当所调用的方法有返回值时，需要将返回值通过 ServerRequest 对象返回给客户端。

(9) 异常处理。当发生异常时，需要将异常信息通过 ServerRequest 对象返回给客户端。

(10) closeAccount()方法的处理过程与 openAccount()方法的处理过程类似。

由上述过程可知，Skeleton 类的构造是比较复杂的，要想真正理解该类的定义，还需要学习“动态接口”一章的内容。

在实际的程序设计过程中，并不需要了解 Skeleton 类的具体内容，只要继承 Skeleton 类来设计服务器程序即可。

5.4.3 Tie 机制方式

上面介绍了通过继承 Skeleton 类来实现分布式对象的方法，但这种方法有一个问题，这就是由于在 Java 语言中只允许单一继承，也就是只允许有一个父类。如果在使用 Skeleton 继承方式时，分布式对象实现本身也需要通过继承其他类来实现的话，这种方式就难以实现了。为了解决这一问题，可以利用 Tie 机制方式，但这种做法使得分布式对象实现的构造变得复杂了。所谓 Tie 机制方式，也就是委托处理方式，这种方式的特点是通过委托处理来实现分布式对象，这样即使需要继承其他的类来实现分布式对象也是可行的。

图 5.3 给出了利用 Tie 机制方式时分布式对象实现的构造。

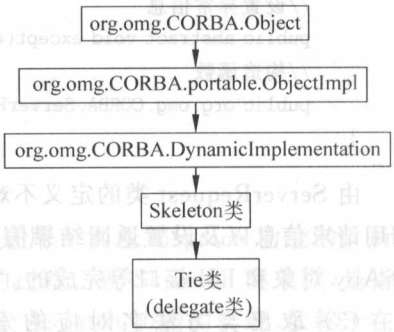

图 5.3 Tie 机制方式的构造

由图 5.3 可以看出，Tie 类本身相当于前述的 Skeleton 继承方式中的分布式对象实现类，Tie 类本身是通过继承 Skeleton 类实现的，而 Tie 类中的具体处理过程则是利用 Delegate 类来完成的。

从图 5.3 中不难看出，在 Tie 机制方式中，为了实现一个分布式对象需要定义两个对象：一个是 Tie 对象，另一个是提供分布式对象中的方法实现的 delegate 对象，即委托对象，这两个对象是通过委托方式结合在一起的。下面具体介绍一下 Tie 机制的实现过程。

1. Tie 类的构成

下面的程序段给出了一种 Tie 类的具体定义。

```
public class _tie_Control extends Bank._ControlImplBase
{
    //代理对象的定义及其访问的方法
    private Bank.ControlOperations _delegate;
    public Bank.ControlOperations _delegate()
    {
        return this._delegate;
    }
    //构造函数
    public _tie_Control(Bank.ControlOperations delegate,java.lang.String name)
    {
        super(name);
        this._delegate = delegate;
    }
    public _tie_Control(Bank.ControlOperations delegate)
    {
        this._delegate = delegate;
    }
```

```
    //openAccount()方法启动请求的代理
    public Bank.Account openAccount(java.lang.String acct,java.lang.String passwd)
                                    throws Bank.ControlPackage.AccountNotExist
    {
        return this._delegate.openAccount(acct,passwd);
    }
    //closeAccount() 方法启动请求的代理
    ⋮
}
```

由_tie_Control类的定义不难看出，Tie类同前面介绍的采用Skeleton继承方式的分布式对象实现的类具有非常相似的构造。

Tie类本身仅提供构造函数和实现由分布式对象接口所定义的操作的方法，而其他由分布式对象实现必须提供的功能则全部由_ControlImplBase类来提供，该类即是前面已介绍的Skeleton类。

与Skeleton继承方式不同的是，在Tie类的构造函数中将委托对象保存在Tie类中，而具体的分布式对象中的方法则是由委托对象中的方法来实现的。

需要注意的是，由分布式对象提供的功能是通过Tie对象由委托对象来实现的，而Tie对象则实现方法启动的发送(dispatch)、分布式对象的语义以及分布式对象的信息保存等功能。

2. 委托对象

委托对象是具体实现分布式对象功能的对象。下面的程序段给出了委托对象的基本构成。

```
public class ControlImplTie implements Bank.ControlOperations
{
    //构造函数
    ControlImplTie(String bank,org.omg.CORBA.BOA boa)
    {
    }
    public Bank.Account openAccount(String acctid,String passwd)
                        throws Bank.ControlPackage.AccountNotExist
    {
        ⋮
    }
    public void closeAccount(String acctid)
    {
        ⋮
    }
}
```

由上述代码不难看出，委托对象只提供构造函数和实现分布式对象功能的方法，委托对象同Skeleton继承方式的分布式对象实现是非常相似的。

从委托类ControlImplTie的定义可以看出，该类没有继承其他类，这样就可以通过类的继承来实现分布式对象的功能，这也是采用Tie机制方式的目的。

另外，从委托类ControlImplTie的定义可以看出，该类实现的接口不是Bank.Control，

而是 Bank. ControlOperation，这是因为在对 IDL 定义进行映射操作以后所产生的接口 Bank. Control 自动继承了 org. omg. CORBA. Object 接口，这样，如果在委托对象中实现接口 Bank. Control，则必须要对分布式对象的语义进行实现，为了避免这一问题，在 IDL 编译时，除了生成 Bank. Control 接口之外，还同时生成了 Bank. ControlOperation 接口，该接口并不继承 org. omg. CORBA. Object 接口。

下面的程序段给出了 Bank. ControlOperation 接口的定义。

```
public interface ControlOperation
{
    public Bank.Account openAccount(java.lang.String acct,java.lang.String passwd)
                        throws Bank.ControlPackage.AccountNotExist;
    public void closeAccount(java.lang.String acct);
}
```

3. Tie 机制方式的使用

上面介绍了 Tie 机制方式的处理过程，下面的程序段给出了如何利用 Tie 机制来实现服务器进程的过程。

```
import org.omg.CORBA.*;
public class ControlServerTie
{
    public static void main(String args[ ])
    {
        try
        {
            //ORB 和 BOA 对象的获取
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            org.omg.CORBA.BOA boa = orb.BOA_init();
            //委托对象的生成
            ControlImplTie control = new ControlImplTie(args[0],boa);
            //Tie 对象的生成
            Bank._tie_Control tie_control = new Bank._tie_Control(control,args[0]);
            //分布式对象的登录
            boa.obj_is_ready(tie_control);
            //接收请求开始
            boa.impl_is_ready();
        }
        catch(… )
        {
            ⋮
        }
    }
}
```

上述程序首先生成委托对象，然后以委托对象的引用为参数生成 Tie 对象（即在 Tie 对象中保存委托对象的引用），然后，将 Tie 对象作为分布式对象登录到对象适配器 OA 中。这样，就可以接收来自客户端的调用请求了。

5.5 分布式对象的生成

所谓分布式对象的生成是指生成分布式对象实现的实例。在Java中，通常是利用new操作来实现的。

由于分布式对象实现的类是通过一系列继承的关系来实现的，因此在生成分布式对象实现的实例时，将要执行各自类的构造函数。

在生成分布式对象时，一般来讲，要进行如下一些操作：

(1) 对分布式对象服务部分的初始化，也就是为分布式对象中的变量等进行初始化。

(2) 对分布式对象语义部分的初始化。如前所述，分布式对象的语义是在org.omg.CORBA.Object接口中定义的，对语义部分进行初始化也就是建立能够对分布式对象进行处理的基础。

(3) 与分布式对象的ID有关的信息设置。该信息的设置主要用于在进行连接操作时能够准确定位相应的分布式对象。比如，在本章的例子中可以看到，在定义分布式对象时，是将BANK传递给分布式对象的构造函数，在该构造函数中，将该名字存放到Skeleton类的相应变量中，这样，在利用obj_is_ready()方法来登录分布式对象时，通过BOA就将其登录到连接服务(Binding Service)中，从而为连接服务打下基础。

5.6 分布式对象的登录

到现在为止，分布式对象已经生成并可以执行了。但是，现在还不能接收来自ORB的请求。对于ORB来讲，它还不认识这个对象。

为了使ORB能够管理分布式对象，就需要将“已经生成新对象并处于接收请求的状态”这一信息通知ORB，这一通知的处理过程被称为分布式对象的登录。

如前面已经看到的那样，在有的CORBA系统中所提供的obj_is_ready()方法就是用于分布式对象登录的。

与obj_is_ready()相对应的deactivate_obj()方法是用于删除已登录的对象，由此方法所指定的分布式对象将从BOA中删除，以便终止对调用请求的接收。

5.7 接收请求开始

分布式对象登录到BOA以后，就可以开始接收来自客户端的请求了，CORBA服务器的处理也就开始了。一般来讲，此时，就进入了等待请求的循环之中。如果这种循环因某种原因结束，也就是服务器进程结束的时候。

在有的CORBA中，调用了impl_is_ready()方法后，就向BOA发出了“服务器的初始化处理已完成，可以进行启动方法的请求处理了”的通知，这时也就进入了接收请求的循环之中。因此，调用了impl_is_ready()方法以后，对请求的处理实际上也就开始了。

到此为止，CORBA服务器的处理也就结束了。

到目前为止，CORBA 服务器的设计都是基于 BOA 来完成的，由于 BOA 的特点，不同系统在实现过程上存在较大差异，在 BOA 的获取、分布式对象的登录以及接收请求开始等函数调用方面都是有差异的。为了解决这一问题，在 CORBA 2.2 以后的版本中提供了 POA。本章的后续部分将介绍 POA 的基本内容。

5.8 POA 基础

POA 为实现 CORBA 服务器提供了非常广阔的选择余地. 如果理解了 POA 的功能，同时，选择了与应用程序特性相吻合的实现策略的话，就能够设计出可扩充性非常好的服务器程序。

5.8.1 POA 中的 CORBA 对象与 Servant 的关系

POA 的主要功能是管理 CORBA 对象与 Servant（服务对象）的生存周期，并将来自客户端的对象调用请求分发给相应的 Servant。这里的 Servant 是指实现 CORBA 对象的 Java 等程序设计语言中的对象。

POA 与 BOA 的最大区别就在于将 CORBA 对象与 Servant 进行完全分离，这种分离提高了服务器应用的可扩展性。如果 CORBA 对象与 Servant 等价的话，则当系统中支持大量的 CORBA 对象时，相同数量的 Servant 也必须保存在内存中。

在 POA 中，CORBA 对象与 Servant 不需要一对一进行连接，通过利用 Servant 池，当客户端的调用请求到来时，从 Servant 池中取出一个 Servant 实例，与被调用的 CORBA 对象进行连接，这种方式可以处理数量巨大的 CORBA 对象。

同时，也可以将所有的 CORBA 对象的处理都委托给一个 Servant 实例来做，这样，可以利用有限的内存来处理大量的 CORBA 对象。

将 CORBA 对象与对应的 Servant 连接之后，就可以处理来自客户端的调用请求了，这一过程被称为激活（activation）。相反，将 CORBA 对象与 Servant 分离的过程被称为去活（deactivation）。

5.8.2 POA 与策略

应用程序开发者所采用的 CORBA 对象与 Servant 的映射方法、对象 ID 的分配方法是使用持续对象还是使用瞬时对象等实现方法都是由 POA 策略来设定的。

如果在程序中不设置 POA 策略的话，就是使用根 POA，即 rootPOA。在 rootPOA 中，已经预先设定了相应的策略（policy），当这一默认的策略不能满足需要时，可以创建在 rootPOA 下设定自己需要策略的 POA。

如图 5.4 所示，可以在 rootPOA 下，层次化地生成 POA 实例。

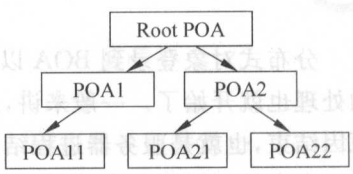

图 5.4 POA 的层次关系

5.8.3 POA的生成

为了生成POA，首先需要生成策略对象(policy object)数组(或称序列)以保存相应的策略。这里，策略对象是通过调用在POA接口中为各个策略定义的“create_策略名”操作来生成的。策略对象数组生成完了之后，以该数组为参数调用其父POA(比如根POA)的create_POA()操作来生成所需的POA。

生成POA及其策略的操作是在PortableServer::POA接口中定义的。下面的程序段给出了PortableServer::POA接口的具体定义。

```
module PortableServer
{
    interface POA
    {
        Exception AdapterAlreadyExists{ }
        Exception AdapterNonExistent{ }
        Exception InvalidPolicy{unsigned short index;}
        //POA的生成
        POA create_POA(in string adapter_name,in POAManager a_POAManager,
              in CORBA::PolicyList policies)raises(AdapterNonExistent,InvalidPolicy);
        //POA的检索
        POA find_POA(in string adapter_name,in boolean activate_it)
                                                raises(AdapterNonExistent);
        //POA的删除
        void destroy(in boolean etherealize_objects,in boolean wait_for_completion);
        //POA策略的生成
        ThreadPolicy create_thread_policy(in ThreadPolicyValue value);
        LifespanPolicy create_lifespan_policy(in LifespanPolicyValue value);
        IdUniquenessPolicy create_id_uniqueness_policy(in IdUniquenessPolicyValue value);
        IdAssignmentPolicy create_id_assignment_policy(in IdAssignmentPolicyValue value);
        ImplicitActivationPolicy create_implicit_activation_policy(
                                                in ImplicitActivationPolicyValue value);
        ServantRetentionPolicy create_servant_retention_policy(
                                                in ServantRetentionPolicyValue value);
        RequestProcessingPolicy create_request_processing_policy(
                                                in RequestProcessingPolicyValue value);
        ⋮
    };
};
```

说明：

(1) 在POA接口中使用的CORBA::PolicyList是在CORBA模块中利用Policy接口定义的序列(sequence)类型。CORBA::Policy接口是包括POA策略在内的CORBA各种策略的基本接口，其定义如下：

```
module CORBA
{
    typedef unsigned long PolicyType;
    interface Policy
```

```
    {
        readonly attribute PolicyType policy_type;
        Policy copy();
        void destroy();
    };
    typedef sequence<Policy>PolicyList;
};
```

(2) POA 接口中的 destroy()方法用于删除已经生成的 POA。

下面的程序段给出了生成 POA 的具体过程。

```
//生成策略数组
Policy accountPolicies[] = new Policy[4];
accountPolicies[0] =
    _rootPoa.create_lifspan_policy(LifespanPolicyValue.PERSISTENT);
accountPolicies[1] =
    _rootPoa.create_id_assignment_policy(IdAssignmentPolicyValue.USER_ID);
accountPolicies[2] =
        _rootPoa.create_implicit_activation_policy(
                ImplicitActivationPolicyValue.NO_IMPLICIT_ACTIVATION);
accountPolicies[3] =
    _rootPoa.creat_thread_policy(ThreadPolicyValue.ORB_CTRL_MODEL);
//生成 POA
try
{
    _accountPoa = _rootPoa.create_POA("Account",null,accountPolicies);
}
catch(AdapterAlreadyExists excep)
{
    System.out.println("This POA have already existed!");
    System.exit(1);
}
catch(InvalidPolicy excep)
{
    System.out.println("The policy is not correct!");
    System.exit(1);
}
```

POA 被创建之后，就可以删除所生成的 Policy 对象。下面的程序段给出了上面创建的 Policy 对象的删除过程。

```
for(int i = 0;i<4;i++)
  accountPolicies[i].destroy();
```

这里，删除 Policy 对象的 destroy()方法是在 Policy 接口中定义的。

5.8.4 POA 策略简介

1. POA 策略一览表

POA 为应用程序提供了许多可以设置的策略，这些策略的正确组合与运用，能够设计

出易于扩充和维护的 CORBA 服务器程序。表 5.1 给出了能够为 POA 设置的 7 种策略。

表 5.1 POA 策略一览表

策 略	取 值	根 POA 的值	默 认 值
ThreadPolicy	ORB_CTRL_MODEL SINGLE_THREAD_MODEL	ORB_CTRL_MODEL	ORB_CTRL_MODEL
LifeSpanPolicy	TRANSIENT PERSISTENT	TRANSIENT	TRANSIENT
IdAssignmentPolicy	SYSTEM_ID USER_ID	SYSTEM_ID	SYSTEM_ID
IdUniquenessPolicy	UNIQUE_ID MULTIPLE_ID	UNIQUE_ID	UNIQUE_ID
ServantRetentionPolicy	RETAIN NON_RETAIN	RETAIN	RETAIN
RequestProcessingPolicy	USE_ACTIVE_OBJECT_ MAP_ONLY USE_SERVANT_MANAGER USE_DEFAULT_SERVANT	USE_ACTIVE_ OBJECT_MAP_ONLY	USE_ACTIVE_ OBJECT_MAP_ONLY
ImplicitiActivationPolicy	IMPLICIT_ACTIVATION NO_IMPLICIT_ACTIVATION	IMPLICIT_ACTIVATION	NO_IMPLICIT_ ACTIVATION

表 5.1 中的策略都是在分别继承了 CORBA::Policy 接口的 IDL 接口中描述的。对于根 POA 来说，其策略值都是预先设置好的，而对于新生成的 POA 来讲，如果没有明确地指定相应的策略值，则使用默认策略值。下面简单介绍每个策略的意义。

2. Threadpolicy

ThreadPolicy，即线程策略，共有两个可选值：ORB_CTRL_MODEL 和 SINGLE_THREAD_MODEL。

(1) ORB_CTRL_MODEL：客户端对该 POA 的对象调用请求可以通过多线程同时进行处理。

(2) SINGLE_THREAD_MODEL：客户端对该 POA 的对象调用请求自动地被顺序化，同时只有一个线程在执行。

下面的程序段给出了 ThreadPolicy 的接口定义。

```
module PortableServer
{
   :
   const CORBA::PolicyType THREAD_POLICY_ID = 16;
   enum ThreadPolicyValue{ORB_CTR_MODEL,SINGLE_THREAD_MODEL};
   interface ThreadPolicy: CORBA::Policy
   {
       readonly attribute ThreadPolicyValue value;
   };
   :
};
```

3. LifespanPolicy

CORBA 对象可以分为两种类型：一种是持续对象，另一种是瞬时对象。持续对象能够在服务器进程结束时继续存在，也就是说，一旦创建了持续 CORBA 对象，即使此后服务器进程结束了，那么重新启动时，客户端仍然可以利用同样的对象引用来继续使用该对象。而瞬时对象，不能够在服务器进程结束时继续存在。

LifespanPolicy，即生命周期策略，共有两个可选值：PERSISTENT 和 TRANSIENT。

(1) PERSISTENT：表示在此 POA 下所生成的对象是持续对象。

(2) TRANSIENT：表示在此 POA 下所生成的对象是瞬时对象。

下面的程序段给出了 LifespanPolicy 的接口定义。

```
module PortableServer
{
    :
    const CORBA::PolicyType LIFESPAN_POLICY_ID = 17;
    enum LifespanPolicyValue{TRANSIENT,PERSISTENT};
    interface LifespanPolicy: CORBA::Policy
    {
        readonly attribute LifespanPolicyValue value;
    };
    :
};
```

4. 有关对象 ID 的策略

1) Object ID 与 Object Key

对象 ID 是在一个 POA 内来唯一确定 CORBA 对象的标识符，是区分由一个 POA 在其自身中所生成的对象，而不是在全局上区分 CORBA 对象的。而在全局上唯一区分一个对象数据的则是被保存在 IOR(Interoperable Object Reference)中的对象键。对象键与对象 ID 的映射方法是由 ORB 的实现来决定的。

2) IdAssignmentPolicy

IdAssignmentPolicy，即 ID 分配策略，共有两个可选值：USER_ID 和 SYSTEM_ID。

(1) USER_ID：在生成对象时，由应用程序来分配对象 ID。

(2) SYSTEM_ID：由 POA 自动地分配唯一的对象 ID。

下面的程序段给出了 IdAssignmentPolicy 的接口定义。

```
module PortableServer
{
    :
    const CORBA::PolicyType ID_ASSIGNMENT_POLICY_ID = 19;
    enum IdAssignmentPolicyValue{USER_ID,SYSTEM_ID};
    interface IdAssignmentPolicy:CORBA::Policy
    {
        readonly attribute IdAssignmentPolicyValue value;
    };
```

```
    ⋮
};
```

3) IdUniquenessPolicy

在 POA 中,CORBA 对象和 Servant 没有必要一一对应,也就是说,一个 Servant 可以同时实现多个 CORBA 对象。设定此项功能的策略是 IdUniquenessPolicy。该策略共有两个可选值:UNIQUE_ID 和 MULTIPLE_ID。

(1) UNIQUE_ID:一个 Servant 同时只能实现一个 CORBA 对象。

(2) MULTIPLE_ID:一个 Servant 可同时实现多个 CORBA 对象。

下面的程序段给出了 IdUniquenessPolicy 的接口定义。

```
module PortableServer
{
    ⋮
    const CORBA::PolicyType ID_UNIQUENESS_POLICY_ID = 18;
    enum IdUniquenessPolicyValue{ UNIQUE_ID,MULTIPLE_ID };
    interface IdUniquenessPolicy:CORBA::Policy
    {
        readonly attribute IdUniquenessPolicyValue value;
    };
    ⋮
};
```

5. ServantRetentionPolicy

将 CORBA 对象映射为 Servant 的最简单的方法是直接利用由 POA 管理的被称为 Active Object Map(活动对象图)的映射表。在该方法中,对于新创建的 CORBA 对象被激活之后,将 CORBA 对象与 Servant 之间的关系被登录在活动对象图中。当来自客户端的对象调用请求到达 POA 之后,POA 在活动对象图中查找与被请求的对象 ID 相对应的 Servant,并据此来转发调用请求。每个 POA 都保存自己的活动对象图。

ServantRetentionPolicy,即 Servant 保持策略,用于设置是否使用活动对象图。共有两个可选值:RETAIN 和 NON_RETAIN。

(1) RETAIN:使用活动对象图。

(2) NON_RETAIN:不使用活动对象图。

如果使用活动对象图,则由于 CORBA 对象与 Servant 的映射是自动进行的,这样服务器程序的实现就比较简单。如果不使用活动对象图,则或者通过创建 Servant 管理器,在程序中明确给出 CORBA 对象与 Servant 的映射关系,或者默认 Servant。

下面的程序段给出了 ServantRetentionPolicy 的接口定义。

```
module PortableServer
{
    ⋮
    const CORBA::PolicyType SERVANT_RETENTION_POLICY_ID = 21;
    enum ServantRetentionPolicyValue{RETAIN,NON_RETAIN};
    interface ServantRetentionPolicy:CORBA::Policy
    {
```

```
        readonly attribute ServantRetentionPolicyValue value;
    };
    :
};
```

6. 有关请求的处理方法的策略

1) RequestProcessingPolicy

RequestProcessingPolicy 策略用于决定采用哪种方式为客户端的调用请求选择 Servant，主要包括利用活动对象图（Active Object Map）、利用 Servant 管理器和默认 Servant 等方式。

RequestProcessingPolicy，即请求处理策略，用于设置对客户端请求的处理方式。共有三个可选值：USE_ACTIVE_OBJECT_MAP_ONLY、USE_DEFAULT_SERVANT 和 USE_SERVANT_MANAGER。

(1) USE_ACTIVE_OBJECT_MAP_ONLY：使用活动对象图。

(2) USE_DEFAULT_SERVANT：默认 Servant。

(3) USE_SERVANT_MANAGER：利用 Servant 管理器。

2) 利用活动对象图

当为 RequestProcessingPolicy 策略指定 USE_ACTIVE_OBJECT_MAP_ONLY 时，对 Servant 的选择就仅使用活动对象图。如果所请求的对象 ID 在活动对象图中不存在的话，则向客户端返回 OBJECT_NOT_EXIST 系统异常。需要注意的是，当选择 USE_ACTIVE_OBJECT_MAP_ONLY 时，就不能为 ServantRetentionPolicy 选择 NON_RETAIN 策略。

3) 利用 Servant 管理器

当为 RequestProcessingPolicy 策略指定 USE_SERVANT_MANAGER 时，利用应用程序开发者所创建的 Servant 管理器来选择 Servant。在这种情况下，根据 ServantRetentionPolicy 所设定的不同的值，有两种情况需要考虑：一是为 ServantRetentionPolicy 策略指定 RETAIN 的情况，二是为 ServantRetentionPolicy 策略指定 NON_RETAIN 的情况。在第一种情况下，POA 根据来自客户端的调用请求，首先查找活动对象图，如果找到所需要的对象 ID，则将调用请求分发给该对象 ID 所对应的 Servant，如果没有找到，则由 Servant 管理器来分配 Servant。这种类型的 Servant 管理器被称为 Servant 激活器。在第二种情况下，则直接由 Servant 管理器来分配 Servant，这种类型的 Servant 管理器被称为 Servant 定位器。

4) 利用默认 Servant

当为 RequestProcessingPolicy 策略指定 USE_DEFAULT_SERVANT 时，表示要利用默认 Servant。在这种情况下，根据 ServantRetentionPolicy 所设定的不同的值，有两种情况需要考虑：一是为 ServantRetentionPolicy 策略指定 RETAIN 的情况，二是为 ServantRetentionPolicy 策略指定 NON_RETAIN 的情况。在第一种情况下，POA 根据来自客户端的调用请求，首先查找活动对象图，如果找到所需要的对象 ID，则将调用请求分发给该对象 ID 所对应的 Servant，如果没有找到，则将调用请求分发给默认 Servant.。在第二种情况下，则直接将调用请求分发给默认 Servant。

下面的程序段给出了 RequestProcessingPolicy 的接口定义。

```
module PortableServer
{
   ⋮
   const CORBA::PolicyType REQUEST_PROCESSING_POLICY_ID = 22;
   enum RequestProcessingPolicyValue{USE_ACTIVE_OBJECT_MAP_ONLY,
                                     USE_DEFAULT_SERVANT,
                                     USE_SERVANT_MANAGER };
   interface RequestProcessingPolicy:CORBA::Policy
   {
      readonly attribute RequestProcessingPolicyValue value;
   };
   ⋮
};
```

7. 有关对象的激活策略

1) 对象的显式激活和隐式激活

所谓对象的激活(activation),是指将 CORBA 对象与 Servant 联系起来,以便能够处理来自客户端的调用请求。相反,如果将 CORBA 对象与 Servant 分离开来,则称为去活(deactivation)。对象的显式激活,是指直接利用 POA 对象中的 activate_object()方法等来激活对象;对象的隐式激活是指在对 Servant 请求对象引用或对象 ID 时才自动地激活对象。

2) ImplicitActivationPolicy

ImplicitActivationPolicy,即隐含激活策略,用于设置是否允许使用隐式激活方式。共有两个可选值:IMPLICIT_ACTIVATION 和 NO_IMPLICIT_ACTIVATION。

(1) IMPLICIT_ACTIVATION:允许使用隐式激活方式。设置这种激活方式的同时,需要将 IdAssignmentPolicy 设置为 SYSTEM_ID 和将 ServantRetentionPolicy 设置为 RETAIN。

(2) NO_IMPLICIT_ACTIVATION:需要利用 POA 的 activate_object()方法或 activate_object_with_id()来显式激活对象。

下面的程序段给出了 ImplicitActivationPolicy 的接口定义。

```
module PortableServer
{
   ⋮
   const CORBA::PolicyType IMPLICIT_ACTIVATION_POLICY_ID = 20;
   enum ImplicitActivationPolicyValue{ IMPLICIT_ACTIVATION,
                                       NO_IMPLICIT_ACTIVATION };
   interface ImplicitActivationPolicy:CORBA::Policy
   {
      readonly attribute ImplicitActivationPolicyValue value;
   };
   ⋮
};
```

5.8.5 POA 管理器

POA 管理器用于管理各 POA 对来自客户端的请求的处理流程。在一个 POA 管理器下可以创建多个 POA。POA 管理器根据其状态来决定如何处理来自客户端的请求，包括将请求加入队列中、分发给目标 POA 以及废弃请求等。

1）POA 管理器的状态

POA 管理器有 4 种状态：HOLDING（保持）、ACTIVE（活动）、DISCARDING（废弃）和 INACTIVE（去活）。这几种状态可以在应用程序中通过调用 POAManager（POA 管理器）接口中的 hold_requests()、activate()、discard_requests()和 deactivate()方法来控制。

（1）HOLDING 状态：POA 管理器是以 HOLDING 状态被创建的，在此状态下，不将来自客户端的对象调用请求发送给 POA，而是保存在队列中。在此期间，服务器程序可以做好准备以处理来自客户端的请求。

（2）ACTIVE 状态：调用 activate()方法可以使 POA 管理器进入 ACTIVE 状态。在此状态下，被保存在队列中的请求以及新到达的请求将被发送给对应的 POA。

（3）DISCARDING 状态：调用 discard_requests()方法可以使 POA 管理器进入 DISCARDING 状态。在此状态下，新到达的请求将被丢弃，并向客户端返回 TRANSIENT 系统异常。

（4）INACTIVE：调用 deactivate()方法可以使 POA 管理器进入 INACTIVE 状态。在此状态下，将拒绝所有来自客户端的新请求。

2）POA 管理器的生成与删除

POA 管理器的生成与删除是在 POA 的生成与删除时隐含进行的。在调用 resolve_initial_reference("rootPOA")方法时，将返回已与 POA 管理器建立联系的根 POA 的引用。这样，通过下述方法就可以获得根 POA 的 POA 管理器 poaMnger：

```
poaMnger = rtPOA.the_POAManager();
```

这里，rtPOA 是根 POA 的对象引用。

以根 POA 的 POA 管理器为参数，通过调用根 POA 的 create_POA()方法，就可以在根 POA 管理器下创建新的 POA。比如：

```
accPOA = rtPOA.create_POA("Acc",POAMnger,accPolicies);
```

这里，accPOA 就是新创建的 POA。

当 POA 管理器下的所有 POA 都被删除时，POA 管理器也将同时被删除。

5.8.6 Servant 管理器

通过利用 Servant 管理器，在应用程序中可以明确地将 CORBA 对象映射为 Servant。Servant 管理器有两种：一种是 Servant 激活器（Servant Activator），另一种是 Servant 定位器（Servant Locator）。

Servant 激活器是与活动对象图并用的，当客户端所要请求的对象在活动对象图中不存在时，则利用 POA 进行调用。在此情况下，一旦 Servant 激活器将对象与 Servant 的映射关

系通知 POA,那么 POA 将把该映射关系登录到活动对象图中,这样,对同一对象的下次调用将直接利用活动关系图。

Servant 定位器是不需要活动关系图而单独使用的。在此情况下,POA 针对每个来自客户端的对象调用请求都要通过调用 Servant 定位器来获取 Servant 的引用。

Servant 激活器和 Servant 定位器是通过在应用程序中实现所给定的接口(ServantActivator 接口和 ServantLocator 接口)并将其实现类的对象登录到 POA 中来实现,这里就不再详细叙述了。

5.8.7 默认 Servant

如果希望对某个 POA 的所有 CORBA 对象的请求都由一个 Servant 来处理的话,就可以使用默认 Servant。要想使用默认 Servant,必须将 IdUniquenessPolicy 策略设置为 MULTIPLE_ID,同时将 RequestProcessingPolicy 策略设置为 USE_DEFAULT_SERVANT。在此设置的基础上,如果 ServantRetensionPolicy 被设置为 RETAIN,则默认 Servant 与活动对象图可以并用;如果 ServantRetensionPolicy 被设置为 NON_RETAIN,则不使用活动对象图,由默认 Servant 处理所有的对象调用请求。

由于在默认 Servant 的实现程序中需要进行与每个 CORBA 对象相关的处理,因此必须要能够获取调用对象的对象 ID,这可以利用 Portable::Current 接口中的 get_object_id()方法。

为了能够利用默认 Servant,首先需要生成默认 Servant 的实例,然后利用 POA 接口中的 set_servant()方法将其登录到具有上述策略的 POA 中。这里需要注意的是,无论是一般 Servant 也好,还是默认 Servant 也好,都是需要用户自己定义的类。

5.8.8 基于 POA 的服务器程序设计

下面以 4.1 节给出的 IDL 定义中的 Account 接口的实现来说明基于 POA 的服务器程序设计过程。

```
public class BankAccServer
{
    private static ORB orbvar;
    private static POA poavar;
    private static POAManager poaMnger;
    private static AccountImplementation accServant;
    private static org.omg.CORBA.Object accRef;
    private static NamingCon text rtNameCtx;
    public static void main( String args[ ])
    {
        org.omg.CORBA.Object tmp;
        //(1) ORB 的初始化
        orbvar = ORB.init(args,null);
        if(orbvar == null)
        {
            System.out.println("Error:ORB initializing. ");
            System.exit(1);
```

```
}
//(2) 获取根 POA 的对象引用
try
{
    tmp = orbvar.resolve_initial_references("RootPOA");
}
catch(…)
{
    ⋮
}
poavar = org.omg.PortableServer.POAHelper.narrow(tmp);
if(poavar == null)
{
    System.out.println("Error:POA initializing. ");
    System.exit(1);
}
//(3) Account 对象的生成
try
{
    accServant = new AccountImplementation();
    poavar.activate_object(accServant);
    accRef = poavar.servant_to_reference(accServant);
}
catch(…)
{
    ⋮
}
//(4) 命名服务对象的获取
try
{
    tmp = orbvar.resolve_initial_references("NameService");
}
catch(…)
{
    ⋮
}
rtNameCtx = NamingContextHelper.narrow(tmp)
if(rtNameCtx == null)
{
    System.out.println("Error: Name Service");
    System.exit(1);
}
//(5) 将分布式对象引用登录到命名服务中
NameComponent[ ] accNameComp = new NameComponent[1];
accNameComp[0] = new NameComponent("Acc","Acc");
try
{
    rtNameCtx.rebind(accNameComp,accRef);
}
catch(…)
{
```

```
            ⋮
        }
        //(6) POA管理器的激活
        poaMnger = poavar.the_POAManager();
        try
        {
            poaMnger.activate();
        }
        catch(…)
        {
            ⋮
        }
        //(7) 等待来自客户端的调用请求
        try
        {
            orbvar.run();
        }
        catch(…)
        {
            ⋮
        }
    }
}
```

说明：

(1) ORB初始化的目的就是获取ORB对象的引用，这是服务器程序设计的第一步。

(2) 根POA对象引用的获取方式与命名服务对象的获取方式是一样，只是在调用resolve_initial_references()方法时需要将RootPOA作为参数。

(3) Account对象的生成同本地对象的生成一样，生成了Account对象之后，需要利用activate_object()方法来激活该对象，同时，还需要利用servant_to_reference()方法来生成对象引用，以便登录到命名服务中。

(4) 利用resolve_initial_reference()方法来获取命名服务对象。

(5) 按照给定的结构将分布式对象引用登录到命名服务中。

(6) 激活POA管理器，以便使其处于活动状态，从而能够将来自客户端的调用请求分发给对应的POA。

(7) 进入等待来自客户端的调用请求的循环之中，这是服务器程序的最后一项工作。

上述程序是属于CORBA服务器进程的程序，作为CORBA服务器程序，还必须包括分布式对象实现，下面给出Account接口的实现类的定义。

```
class AccountImplementation extends AccountPOA
{
    private int balanceVar;
    public AccountImplementation()
    {
        balanceVar = 0;
    }
    public int Deposit(int amount)
```

```
    {
        balanceVar += amount;
        return balanceVar;
    }
    public Withdraw(int amount)
    {
        balanceVar-= amount;
        return balanceVar;
    }
    public int balance()
    {
        return balanceVar;
    }
}
```

说明：

(1) 作为基类的 AccountPOA 是 Account 接口经编译之后所生成的 Skeleton 类。

(2) 类中定义的方法名、参数类型以及属性名等必须和 IDL 接口中定义的相同。

第6章 动态接口

在前几章中介绍了CORBA程序设计过程，其中，客户端程序是利用Stub类来实现的，服务器程序是利用Skeleton类实现的。采用这种方式进行程序设计，很多复杂的程序设计细节都被Stub类和Skeleton类屏蔽掉了，程序设计人员就如同实现本地对象一样来实现分布式对象。这种程序设计方法被称为静态程序设计。利用静态程序设计方法能够满足绝大部分应用程序的开发。如果希望设计比较复杂的CORBA应用系统，比如CORBA防火墙等，单纯采用静态程序设计方式就很难实现了，这时就需要采用动态程序设计方式。本章主要介绍与动态程序设计有关的CORBA基本内容，主要包括接口仓库(Interface Repository，IFR)、动态启动接口(Dynamic Invocation Interface，DII)和动态骨架接口(Dynamic Skeleton Interface，DSI)等。

6.1 通用伪接口的定义

本节主要介绍在动态程序设计过程中需要使用的一些通用伪接口，主要包括TypeCode、NamedValue、NVList以及ORB接口等。

6.1.1 TypeCode接口

TypeCode是用于表示IDL类型的伪接口。TypeCode伪接口在接口仓库(IFR)、动态启动接口(DII)、动态骨架接口(DSI)和Any类型的处理中等都要使用到。

1. TypeCode接口定义

在TypeCode接口定义中，需要使用TCKind枚举类型，其定义如下：

```
module CORBA
{
    enum TCKind
    {
        tk_null,tk_void,
        tk_short,tk_long,tk_ushort,tk_ulong,
        tk_float,tk_double,tk_boolean,tk_char,
        tk_octet,tk_any,tk_TypeCode,tk_Principal,tk_objref,
        tk_struct,tk_union,tk_enum,tk_string,
        tk_sequence,tk_array,tk_alias,tk_except,
```

```
        tk_longlong,tk_ulonglong,tk_longdouble,
        tk_wchar,tk_wstring,tk_fixed
        ⋮
    };
    ⋮
};
```

说明：在 TCKind 枚举类型定义中给出了表示各种 IDL 类型的枚举常量的定义，比如 tk_short 用于表示 short 类型，tk_struct 用于表示 struct 类型等。

下面给出 TypeCode 伪接口所对应的 Java 语言映射结果。

```
package org.omg.CORBA;
public abstract class TypeCode extends org.omg.CORBA.portable.IDLEntity
{

    //(1) 所有的 TypeCode 都能使用的方法
    public abstract boolean equal(TypeCode tc);
    public abstract TCKind kind();

    //(2) tk_objref、tk_struct、tk_union、tk_enum、tk_alias、tk_except 等
    //TypeCode 能够使用的方法
    //返回由 TypeCode 所表示的数据类型的仓库 ID
    public abstract String id()throws TypeCodePackage.BadKind;
    //返回由 IDL 定义的用户自定义类型的名字
    public abstract String name()throws TypeCodePackage.BadKind;

    //tk_struct、tk_union、tk_enum、tk_except 等 TypeCode 能够使用的方法
    //返回成员个数
    public abstract int member_count()throws TypeCodePackage.BadKind;
    //返回成员名字
    public abstract String member_name(int index)
            throws TypeCodePackage.BadKind,TypeCodePackage.Bounds;
    //tk_struct、tk_union、tk_except 等 TypeCode 能够使用的方法
    //返回成员类型
    public abstract TypeCode member_type(in index)
            throws TypeCodePackage.BadKind,TypeCodePackage.Bounds;
    //tk_union 等 TypeCode 能够使用的方法
    public abstract any member_label(int index)
                throws TypeCodePackage.BadKind,TypeCodePackage.Bounds;
    public abstract TypeCode discriminator _type()
                                        throws TypeCodePackage.BadKind;
    public abstract int default_index() throws TypeCodePackage.BadKind;

    //tk_string、tk_sequence、tk_array 等 TypeCode 能够使用的方法
    public abstract int length() throws TypeCodePackage.BadKind;
    //tk_sequence、tk_array、tk_alias 等 TypeCode 能够使用的方法
    public abstract TypeCode content_type() throws TypeCodePackage.BadKind;

    // tk_fixed 等 TypeCode 能够使用的方法
    public abstract short fixed_digits()throws TypeCodePackage.BadKind;
```

```
    public abstract short fixed_scale()throws TypeCodePackage.BadKind;
    :
};
```

说明：

(1) TypeCode 接口定义是由所有数据类型的通用部分与各数据类型的专用部分组成的。通用部分包括 TypeCode 的比较操作(equal)与返回当前 TypeCode 所对应的 TCKind 中的枚举量(Kind)。

(2) 与各数据类型相关的操作，我们以结构为例进行简单说明。

对于结构类型的 TypeCode 能够引用的操作可以分为两种：一种是对用户自定义类型通用的操作，即 id()和 name()。id()操作用于返回由 TypeCode 所表现的类型的仓库 ID。所谓仓库 ID，是指系统自动为 IDL 定义的数据类型所分配的标识符，这一点在下面将详细介绍。name()操作用于返回由 TypeCode 所表现的类型的名字。

另一种是访问结构成员的操作。为了访问有关这些成员的信息，定义了三个操作：member_count()、member_name()和 member_type()，它们分别用于返回成员个数、成员的名字和成员类型。

下面将介绍在程序设计过程中如何能够获取所需要的 TypeCode 对象。

2. 基本类型 TypeCode 的获取

为了获取基本类型的 TypeCode，可以利用 ORB 接口中定义的 get_primitive_tc()方法，该方法的 Java 映射结果如下：

```
package org.omg.CORBA;
public abstract class ORB
{
    public abstract TypeCode get_primitive_tc(TCKind tcKind);
    :
}
```

3. 用户定义类型 TypeCode 的生成

由 TypeCode 接口的定义不难看出，在 TypeCode 接口中只定义了取出 TypeCode 信息的操作，而没有定义生成 TypeCode 信息的操作，生成 TypeCode 的操作是在 ORB 接口中定义的。

下面的程序段给出了用户定义类型的 TypeCode 生成方法的 Java 映射结果。

```
package org.omg.CORBA;
public abstract class ORB
{
    public abstract TypeCode create_struct_tc(String id,String name,
                                              StructMember[ ] member);
    public abstract TypeCode create_exception_tc(String id,String name,
                                              StructMember[ ] member);
    public abstract TypeCode create_interface_tc(String id,String name);
    :
```

```
}
```

其中，StructMember 的 IDL 定义如下：

```
module CORBA
{
    struct StructMember
    {
        Identifier name;
        TypeCode type;
        IDLType type_def;
    };
    ⋮
};
```

说明：为了生成用户定义类型的 TypeCode，可以利用 ORB 接口中定义的操作。这里，create_struct_tc()用于生成结构类型的 TypeCode，create_exception_tc()操作用于生成异常类型的 TypeCode，create_interface_tc()操作用于生成接口的 TypeCode。

TypeCode 对象在使用 any 类型变量以及利用后续章节介绍的动态接口进行程序设计时都要经常使用。

6.1.2 NamedValue 接口

NamedValue 是用于保存名字和值的对应关系的伪接口，同时还包括一个 flags 成员。NamedValue 主要用于在动态接口 DII 和 DSI 中描述操作（方法）的参数类型，同时还可用于描述 Context 对象的属性。

1. NamedValue 的接口定义

下面的程序段给出了 NamedValue 接口的 Java 映射结果。

```
package org.omg.CORBA;
public interface ARG_IN
{
    public static final int value = 1;
}
public interface ARG_OUT
{
    public static final int value = 2;
}
public interface ARG_INOUT
{
    public static final int value = 3;
}
public interface CTX_RESTRICT_SCORE
{
    public static final int value = 15;
}

public abstract class NamedValue
```

```
{
    public abstract String name();          //名字
    public abstract Any value();            //值
    public abstract int flags();            //标志. IN, INOUT 和 OUT 等
}
```

说明：Name()方法和 value()方法分别用于返回被保存在 NamedValue 中的名字及其值，其中，值被保存在 any 类型的对象中。flags()方法用于返回标志，其取值为 ARG_IN. value、ARG_OUT. value、ARG_INOUT. value 和 CTX_RESTRICT_SCOPE. value。

2. NamedValue 对象的生成

为了生成 NamedValue 对象，可以利用 ORB 接口中定义的 create_named_value()方法，该方法的 Java 映射结果如下：

```
package org.omg.CORBA;
public abstract class ORB
{
    public abstract NamedValue create_named_value(String name,Any value,
                                                  int flags);
    ⋮
}
```

6.1.3 NVList 接口

NVList 是用于保存名字及其值的列表的伪接口。与 NamedValue 一样，NVList 也是主要用于在动态接口 DII 和 DSI 中描述操作(方法)的参数类型，同时还可用于描述 Context 对象的属性。

1. NVList 的接口定义

下面的程序段给出了 NVList 接口的 Java 映射结果。

```
package org.omg.CORBA;
public abstract class NVList
{
    public abstract int count();
    public abstract NamedValue add(int flags);
    public abstract NamedValue add_item(String item_name,int flags);
    public abstract NamedValue add_value(String item_name,Any val,int flags);
    public abstract NamedValue item(int index)throws org.omg.CORBA.Bounds
    public abstract void remove(int index)throws org.omg.CORBA.Bounds;
}
```

说明：为了向 NVList 对象中追加名字、值和标志，可以利用 add_value()方法。

2. NVList 的生成

为了创建 NVList 列表，可以利用在 ORB 接口中定义的 create_list()方法或 create_operation_list()方法。这些方法的 Java 映射结果如下：

```
package org.omg.CORBA;
public abstract class ORB
{
    public abstract NVList create_list(int count);
    public abstract NVList create_operation_list(OperationDef oper);
    ⋮
}
```

说明：create()方法用于返回一个空的 NVList，参数 count 被用于空间分配的提示性信息。create_operation_list()方法用于返回以参数 OperationDef 为基础所生成的 NVList，主要用于对方法的动态调用处理。

6.2 Any 类型数据的处理

在前面已提到过，所谓 Any 类型就是能存放任何类型数据的数据类型。也就是说，利用 Any 定义的变量，既能存放整型数、字符串，又能存放结构和对象等。这可能给我们这样一种感觉，事先不用对参数的类型进行设计，只要直接利用 Any 类型即可。实际上，这是不行的，原因就在于一是它违反了面向对象程序设计的基本原理，另一个是利用 Any 类型需要比较复杂的接口，也就是说，使用 Any 类型比使用其他类型要复杂得多。因此，通常在 CORBA 程序设计中，尽量不利用 Any 类型为好。

Any 类型主要用在本章要介绍的 IFR（接口仓库）、DII（动态启动接口）和 DSI（动态骨架接口）中。

在 Java 语言映射中，Any 都是作为类来实现的。然而，不同的语言映射所生成的类的定义内容却大不相同。

从 CORBA V2.2 开始，为了消除语言映射的差异，提供了 DynAny（Dynamic Any）接口，以便为操作 Any 类型提供标准的方法。只要实现这一接口，就可以不依赖于语言映射来实现 Any 程序设计。

6.2.1 Any 的功能与数据构造

为了操作 Any 数据类型，至少需要以下几种功能：

（1）将数据存入 Any 的功能。

（2）从 Any 取出数据的功能。

（3）能够对保存在 Any 中的数据的类型信息进行操作的功能。

为了存取基本类型的数据，在 Any 中需分别提供存取这些数据的方法，而对于用户定义类型的数据存取方法，则是在编译 IDL 时与 Stub 和 Skeleton 类同时产生的。

由于 Any 中能够存放任何类型的数据，因此，必须要知道所存放数据的类型。这样，Any 类型在保存数据本身的同时，还必须要保存所存放的数据的类型信息。在访问 Any 之前，先取出其类型信息，在确定了类型之后，再访问数据；相反，在将数据存入 Any 时，需要设置其类型信息。而对数据类型信息进行处理的标准功能则是由 TypeCode 接口提供的。

6.2.2 Any 类型的 Java 映射

IDL 的 Any 类型被映射为 Java 语言的 org. omg. CORBA. Any 类。下面的程序段给出了 Any 类型的 Java 映射结果。

```
package org.omg.CORBA;
abstract public class Any
{
    //(1) 比较操作
    abstract public boolean equal(Any a);
    //(2) 对 TypeCode 的访问
    abstract public TypeCode type();
    abstract public void type(TypeCode t);
    //(3) 基本类型数据的存取
    abstract public short extract_short();
    abstract public void insert_short(short s);
    abstract public short extract_ushort()throws org.omg.CORBA.BAD_OPERATION;
    abstract public void insert_ushort(short s);
    //其他类型的类似操作
    ⋮
};
```

说明：

(1) equal()方法用于比较入口参数的 Any 类型的值与当前 Any 对象中的值是否相同。

(2) 这两个 type()方法分别用于取出和保存 TypeCode 对象。

(3) 这些方法用于对基本类型数据的存取。需要注意的是，这些类型的分类是按 IDL 的类型来进行的，而参数的类型则是由 Java 的类型来决定的。例如，在 IDL 中，short 和 unsigned short 是两个不同的类型，而在 Java 中却没有 unsigned short 类型，这样，用于处理 unsigned short 的方法 extract_ushort()和 insert_ushort()是存在的，但其参数却是使用 Java 的 short。

6.2.3 Any 对象的生成

由于 Any 类型被映射为 Java 语言的抽象 Any 类，所以不能直接生成 Any 类的实例。为了生成 Any 类型的实例(对象)，必须要利用 ORB 接口中定义的 create_any()方法。下面的程序段给出了 create_any()方法的定义方式。

```
package org.omg.CORBA;
public abstract class ORB
{
    public abstract Any create_any();
    ⋮
}
```

6.2.4 Any 对基本类型数据的存取

前面对 Any 接口以及 Any 实例的生成方法进行了说明，在本节将介绍利用 Any 对象

来存取基本类型数据的过程。下面的程序段给出了将基本类型的数据存入 Any 或从 Any 取出的过程。

```
//Any变量的生成
Any any_var = org.omg.ORB.init().create_any();
short short_var1 = 100;
//将 short 数据存入 Any
any_var.insert_short(short_var1);
//从 Any 中取出 short 数据
if(any_var.kind() == TCKind.tk_short)
{
    short short_var2 = any_var.extract_short();
}
```

说明：在利用 Any 来存取基本类型的数据时，只要直接利用 Any 中的方法即可。需要注意的是，在取数据之前，应先检查类型是否正确，也就是要保证 Any 中所保存的数据类型应该与所调用的方法相一致，否则将发生异常。

6.2.5 Any 对用户定义类型数据的存取

在前面给出的 Any 类的定义中，并没有给出对用户自定义类型的处理方法，这是因为用户自定义类型是随着 IDL 的定义动态生成的，而不可能事先定义好。在 Java 映射中，支持用户自定义类型的方法，是按照类型的不同由相应的 Helper 类来提供的。

例如，下面是利用 IDL 定义的结构类型：

```
struct StructDef
{
    string  name;
    short   age;
};
```

前已述及，在进行 Java 映射时，每个用户定义类型都被映射为与其对应的 Helper 类。对于 StructDef 结构类型来讲，其 Helper 类名为 StructDefHelper。下面的程序段给出了在 StructDefHelper 类中定义的与 Any 操作有关的方法。

```
public class StructDefHelper
{
    //与 Any 操作有关的方法
    public static void insert(org.omg.CORBA.Ang a,StructDef us){ };
    public static StructDef extract(org.omg.CORBA.Any a){ };
    public static org.omg.corba.TypeCode type(){ } ;
    ⋮
}
```

由此可知，利用 Any 对象和 Helper 类就可以实现对用户定义类型数据的 Any 存取操作，下面的程序段给出了这一处理过程。

```
//Any变量的生成
ORB orb_var =  org.omg.ORB.init();
Any any_var =  orb_var.create_any();
```

```
StructDef str_var1;
str_var1.name = "Nothing";
str_var1.age = 50;
//将 str_var1 数据存入 Any 中
StructDefHelper.insert(any_var,str_var1);
//从 Any 中取出 StructDef 类型数据
if ((any_var.type()).equal(StructDefHelper.type()))
{
    StructDef str_var2 = StructDefHelper.extract(any_var);
}
```

说明：将用户定义类型数据存入 Any 或从 Any 中取出，是通过利用相应 Helper 类中的方法来实现的。存入操作利用 insert()方法，取出操作则是利用 extract()方法。

6.2.6 DynAny 接口

在 Java 映射的规定中，利用 Any 类来处理用户定义类型时，是利用编译该类型的 IDL 所生成的 Helper 类来实现的，然而，如果事先不了解要处理数据的类型信息的话，就不能向 Any 变量存取用户定义的类型数据。同时，由于需要利用 Helper 类来实现用户定义类型数据的 Any 存取，因此，必须预先对 IDL 定义进行编译，这对于利用 DII 和 DSI 进行动态程序设计就不一定适合。为了解决这一问题，在 CORBA 2.2 版本中提供了 DynAny 接口。利用该接口，就可以不需要 Helper 类这样的静态信息，而可以完全动态地从 Any 取出数据或向 Any 中插入数据。有关 DynAny 接口的具体内容不再详细叙述。

6.3 接口仓库

在前面的介绍中多次提到接口仓库。在 CORBA 接口仓库中，是以 CORBA 对象的形式来保存 IDL 接口信息中的各个元素。接口仓库中的信息一般是在编译 IDL 时生成的。

例如，由 module 定义的信息，在接口仓库中是作为 ModuleDef 接口的对象来表示的，同样，由 interface 定义的信息是作为 InterfaceDef 接口的对象形式来表示的，由操作定义的信息是作为 OperationDef 接口的对象形式来表示的，等等。这些对象的管理方式是以表示接口仓库自身的 Repository 对象为根而形成的一种树型结构。

一般来讲，不利用 Stub 和 Skeleton 而使用 DII 和 DSI 来设计 CORBA 客户端和服务器时需要使用接口仓库。

6.3.1 接口仓库的构造

由图 6.1 可知，接口仓库的内部构造与 IDL 定义的构造完全对应，由 IDL 定义的各种元素在接口仓库中是以对象的形式被管理起来的。

接口仓库中的最上层对象是 Repository。在 Repository 仓库中能够存放的对象包括 ConstantDef、TypedefDef、ExceptionDef，InterfaceDef 和 ModuleDef 等。这些对象的意义由其名字就能体现出来。

在接口仓库中，能够保存其他对象的对象被称为容器（Container）对象。比如，

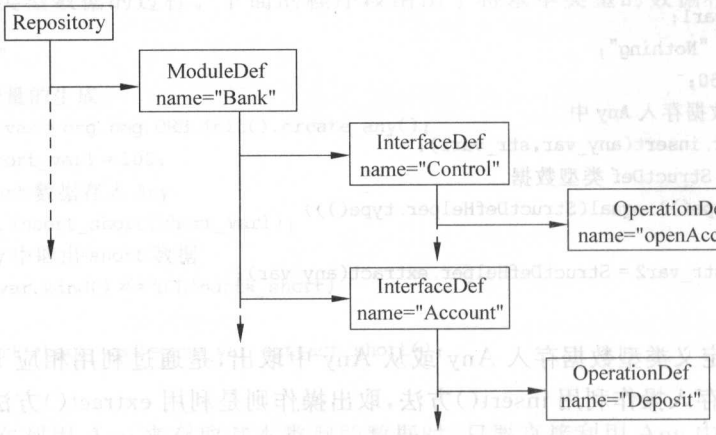

图 6.1 接口仓库的构造

ModuleDef 对象本身就是容器对象，它可以存放 ConstantDef、TypedefDef、ExceptionDef、InterfaceDef 和 ModuleDef 等各个对象。

InterfaceDef 对象本身也是容器对象，它可以存放 ConstantDef、TypedefDef、ExceptionDef、AttributeDef 和 OperationDef 等各个对象。

为了对接口仓库进行访问，首先必须以某种方法获取上述某个对象的引用，具体做法将在下面进行叙述。

6.3.2 接口仓库的接口

在 InterfaceDef 等接口仓库对象的接口中，定义了读取被保存在接口仓库中的信息的方法和更新接口仓库信息的方法。在本节中主要介绍作为父接口使用的 IRObject 接口、Container 接口、Contained 接口和 IDLType 接口以及 InterfaceDef 接口、OperationDef 接口和 AttributeDef 接口等。

1. IRObject 接口

IRObject(Interface Repository Object)接口是接口仓库的所有对象共同具有的接口，其定义如下：

```
module CORBA
{
    interface IRObject
    {
        readonly attribute DefinitionKind def_kind;
        void destroy();
    };
    ⋮
};
```

说明：IRObject 接口比较简单，def_kind 属性用于访问当前对象的类型值；destroy() 方法用于在接口仓库中删除当前对象。如果当前对象是 Container 对象，则执行 destroy()

方法将删除所有属于该 Container 对象的 Contained 对象。

在 IRObject 接口中使用了 DefinitionKind 类型，该类型是由各接口仓库对象的接口所对应的枚举量所组成的枚举类型。其定义如下：

```
module CORBA
{
    typedef string Identifier;
    typedef string ScopedName;
    typedef string RepositoryID;
    enum DefinitionKind
    {
        dk_none,dk_all,
        dk_Attribute,dk_Constant,
        dk_Exception,dk_Interface,
        dk_Module,dk_Operation,dk_Typedef,
        dk_Alias,dk_Struct,dk_Union,dk_Enum,
        dk_Primitive,dk_String,dk_Sequence,dk_Array,
        dk_Repository,dk_Wstring,dk_Fixed
        :
    };
    :
};
```

说明：DefinitionKind 枚举类型主要用于在进行仓库对象检索时指定被检索对象的类型。具体用法将在后面需要的地方进行说明。

2. Contained 接口

在接口仓库的层次构造中，Contained 接口用于表示被其他对象所包含的对象的基本接口，即凡是被包含在其他对象中的对象，其接口都要继承 Contained 接口。比如，OperationDef 对象和 AttributeDef 对象都包含在 InterfaceDef 对象中，因此，这两个对象的接口都要继承 Contained 接口。Contained 接口本身要继承 IRObject 接口。

Contained 接口的定义如下：

```
module CORBA
{
    typedef string VersionSpec;
    interface Contained:IRObject
    {
        attribute RepositoryID id;
        attribute Identifier name;
        attribute VersionSpec version;
        readonly attribute Container defined_in;
        readonly attribute ScopedName absolute_name;
        readonly attribute Repository containing_repository;
        struct Description
        {
            DefinitionKind kind;
            any value;
```

```
        }
        Description describe();
        void move( in Container new_container,in Identifier new_name,
                                              in VersionSpec new_version);
    };
    :
};
```

说明：Contained 接口主要由这样几种信息构成：一是提供自身信息的 id、name、version 属性和 describe()方法，二是提供当前对象所属 Container 对象信息的 defined_in、absolute_name、containing_ repository 属性以及用于改变所属 Container 对象的 move()方法构成的。

3. Container 接口

Container 接口在接口仓库的层次构造中，用于表示包含其他对象的容器对象的父接口，也就是说这些接口都要继承 Container 接口。例如，由于 InterfaceDef 对象可以包含 OperationDef 对象和 AttributeDef 对象等，因此，InterfaceDef 接口需要继承 Container 接口。

Container 接口的定义如下：

```
module CORBA
{
    typedef sequence <Contained> ContainedSeq;
    interface Container:IRObject
    {
        Contained lookup(in ScopedName search_name);
        ContainedSeq contents(in DefinitionKind limit_type,
                              in boolean exclude_inherited);
        ContainedSeq lookup_name(in Identifier search_name,
                                 in long levels_to_search,
                                 in DefinitionKind limit_type,
                                 in boolean exclude_inherited);
        :
    };
    :
};
```

说明：lookup()方法用于检索 Container 属下的由参数所指定的 IDL 元素，并返回其接口仓库对象引用。ScopedName 类型只是利用 typedef 定义的 string 类型，由其所指定的名字既可以是相对域名（比如，Deposit 或 Account::Deposit 等），也可以是绝对域名（比如，::Bank::Account::Deposit 等）。该方法的返回值为 Contained 类型的对象引用。

Lookup_name()方法的功能和 lookup()的功能相似，也是用于检索 Container 属下的接口仓库对象，但可以比 lookup()方法指定更详细的检索条件，并且可以检索多个对象。该方法的第一个参数用于指定作为检索对象的 IDL 元素的名字（Identifier 类型就是 string 类型），第二个参数用于说明是只检索本 Container 对象下的元素（=1），还是以递归的方式检索其下所有 Container 下的元素（=-1）。第三个参数用于指定要检索哪种类型的接口

仓库对象。比如，如果要检索 OperationDef 对象，则只要指定 DefinitionKind. dk_Operation 即可。如果要检索所有类型的接口仓库对象，则要指定 DefinitionKind. dk_all。第四个参数用于说明是否将被继承的 IDL 元素排除在检索对象之外。假如 nextAccount 接口继承了 Account 接口，同时，在 nextAccount 接口的 InterfaceDef 对象上调用 lookup_name()方法，这时，如果第四个参数被指定为 true，则只检索由 nextAccount 接口直接定义的 IDL 元素。如果指定为 false，则 Account 接口中定义的元素也将作为检索对象。

4. IDLType 接口

IDLType 接口是表示 IDL 类型的接口仓库接口的父接口。比如，由于 IDL 的 interface 和用户定义类型都是 IDL 类型，因此，与其对应的 InterfaceDef 接口和 TypeDef 接口都要继承 IDLType 接口。但是，OperationDef 接口和 AttributeDef 接口不需要继承 IDLType 接口，这是由于操作和属性并不是 IDL 类型的原因。

IDLType 接口的定义如下：

```
module CORBA
{
    interface IDLType:IRObject
    {
        readonly attribute TypeCode type;
    };
     ⋮
};
```

说明：在 IDLType 接口中只定义了 type 属性，用于获取当前对象的 TypeCode。

5. Repository 接口

作为接口仓库层次结构中的最顶层(即树根)，Repository 接口只继承 Container 接口。其定义如下：

```
module CORBA
{
    interface Repository:Container
    {
        Contained lookup_id(in RepositoryID search_id);
         ⋮
    };
     ⋮
};
```

说明：Repository 接口中定义的一个重要函数是 lookup_id()，该函数用于检索与入口参数的仓库 ID 所对应的接口仓库对象。

6. InterfaceDef 接口

InterfaceDef 接口的对象用于管理接口定义的信息。InterfaceDef 接口继承了 Container 接口、Contained 接口和 IDLType 接口。

InterfaceDef 接口的定义如下：

```
module CORBA
{
    interface InterfaceDef:Container,Contained,IDLType
    {
        struct FullInterfaceDescription
        {
            Identifier name;
            RepositoryId id;
            RepositoryId defined_in;
            VersionSpec version;
            OpDescriptionSeq operations;
            AttrDescriptionSeq attributes;
            RepositoryIdSeq base_interfaces;
            TypeCode type;
            boolean is_abstract;
        };
        FullInterfaceDescription describe_interface();
        AttributeDef create_attribute(in RepositoryId id,in Identifier name,
                                      in VersionSpec version,in IDLType type,
                                      in AttributeMode mode);
        OperationDef create_operation(in RepositoryId id,in Identifier name,
                    in VersionSpec version,in IDLType result,
                    in OperationMode mode,in ParDescriptionSeq params,
                    in ExceptionDefSeq exception,in ContextIdSeq contexts);
    };
    :
};
```

说明：在 InterfaceDef 接口中定义的 base_interfaces 属性用于保存该接口的所有父接口的 InterfaceDef 序列；describe_interface()操作用于读取接口的详细信息；create_attribute()操作和 create_operation()操作分别用于将属性信息对象和操作信息对象保存到当前接口信息对象中。

7. OperationDef 接口

OperationDef 接口的对象用于管理操作定义的信息。OperationDef 接口继承了 Contained 接口。

OperationDef 接口的定义如下：

```
module CORBA
{
    enum OperationMode{OP_NORMAL,OP_ONEWAY};
    enum ParameterMode{PARAM_IN,PARAM_OUT,PARAM_INOUT};
    struct ParameterDescription
    {
        Identifier name;
        TypeCode type;
        IDLType type_def;
```

```
        ParameterMode mode;
    };
    typedef sequence <ParameterDescription> ParDescriptionSeq;
    typedef Identifier ContextIdentifier;
    typedef sequence <ContextIdentifier> ContextIdSeq;
    typedef sequence <ExceptionDef> ExceptionDefSeq;
    typedef sequence <ExceptionDescription> ExcDescriptionSeq;

    interface OperationDef:Contained
    {
        readonly attribute TypeCode result;
        attribute IDLType result_def;
        attribute ParDescriptionSeq params;
        attribute OperationMode mode;
        attribute ContextIdSeq contexts;
        attribute ExceptionDefSeq exceptions;
    };
    ⋮
};
```

说明：在 OperationDef 接口中，定义了访问操作定义信息的 result、params 和 exceptions 等属性。result 属性用于获取表示操作的返回值类型的 TypeCode；params 属性是保存各参数的详细信息的 ParameterDescription 结构序列；exceptions 属性表示为该操作定义的异常的 ExceptionDef 序列。

8. AttributeDef 接口

AttributeDef 接口的对象用于管理属性定义的信息。AttributeDef 接口继承了 Contained 接口。

AttributeDef 接口的定义如下：

```
module CORBA
{
    enum AttributeMode{ATTR_NORMAL,ATTR_READONLY};

    interface AttributeDef:Contained
    {
        readonly attribute TypeCode type;
        attribute IDLType type_def;
        attribute AttributeMode mode;
    };
    ⋮
};
```

说明：在 AttributeDef 接口中定义了访问属性信息的 type、type_def 和 mode 属性。type 属性是表示属性类型的 TypeCode；type_def 属性是表示该属性的类型在接口仓库内的对象引用；mode 属性是表示该属性是否是只读(readonly)属性。

6.3.3 对接口仓库的访问

为了访问接口仓库，必须首先获取被保存在仓库内的某个对象的引用。其中最有代表

性的方法就是获取作为树型结构的根的 Repository 对象的引用。还有一种方法就是根据分布式对象引用来获取与该对象接口相对应的 InterfaceDef 对象的引用。

1. Repository 对象引用的获取

Repository 作为接口仓库的初始对象引用，其获取方法可以利用在 ORB 接口中定义的 resolve_initial_references()操作，这时需要传递的参数是 InterfaceRepository。下面的程序段给出了这一方法的实现过程。

```
org.omg.CORBA.Repository repo_var;
org.omg.CORBA.Object tmp_var;
ORB orb_var = org.omg.ORB.init();
try
{
    tmp_var = orb_var.resolve_initial_references("InterfaceRepository");
}
catch(…)
{
    ⋮
}
repo_var = org.omg.CORBA.RepositoryHelper.narrow(tmp_var);
⋮
```

说明：在获取了 Repository 对象引用之后，就可以通过调用其方法来遍历接口仓库的树型结构，以获取所需要的接口信息。

2. InterfaceDef 对象引用的获取

为了利用分布式对象引用来获取与其对应的对象的 InterfaceDef 对象引用，需要利用在 org.omg.CORBA.Object 接口中定义的_get_interface_def()方法。该方法返回 org.omg.CORBA.Object 类型的 InterfaceDef 对象引用。下面的程序段给出了这一方法的实现过程。

```
org.omg.CORBA.Object itf_var = objRef._get_interface_def();
if(itf_var == null)
{
    ⋮
}
org.omg.CORBA.InterfaceDef itfDef =
                    org.omg.CORBA.InterfaceDefHelper.narrow(itf_var);
⋮
```

说明：这里的 objRef 是分布式对象引用。如果所获取的 itf_var 为空，则表示相应对象的接口定义信息并没有登录到接口仓库中。

在获得了 InterfaceDef 对象引用之后，就可以通过利用 interfaceDef 接口中定义的方法来获取与接口有关的各种信息。

3. 对接口仓库的漫游操作

由上述过程获取了相关对象的引用之后，就可以利用获取的对象引用来对接口仓库进

行漫游操作。由上述可知，用于漫游的操作在 Repository 接口、Container 接口和 Contained 接口中进行了定义。下面对其进行简单的总结。

1）在 Repository 接口中提供的漫游操作

在 Repository 接口中提供了根据仓库 ID 进行接口仓库对象检索的 lookup_id()方法。

2）在 Container 接口中提供的漫游操作

在 Container 接口中提供的漫游操作包括只根据名字进行检索的 lookup()方法、除了名字之外还加有其他条件进行检索的 lookup_name()方法和对同类型的对象都进行检索的 Contents()方法。

3）在 Contained 接口中提供的漫游操作

在 Contained 接口中提供的漫游操作包括保存 Container 对象的 defined_in 属性和保存 Repository 对象的 containing_repository 属性。

当利用上述方法获取了所需要的对象之后，一般就可以利用 describe()方法来获取相应的信息。

下面的程序段给出了访问接口仓库的基本过程。

```
//(1) 初始 InterfaceDef 对象引用的获取
org.omg.CORBA.Object itf_var = conRef._get_interface_def();
if(itf_var == null)
{
    ⋮
}
org.omg.CORBA.InterfaceDef itfDef =
                    org.omg.CORBA.InterfaceDefHelper.narrow(itf_var);
//(2) 根据名字检索相应的对象
org.omg.CORBA.OperationDef openAccDef =
  org.omg.CORBA.OperationDefHelper.narrow(itfDef.lookup("openAccount"));
//(3) 获取所需对象的信息
org.omg.CORBA.OperationDescription openAccDesc =
  org.omg.CORBA.OperationDescriptionHelper.narrow(openAccDef.describe());
⋮
```

说明：

(1) 根据分布式对象引用来获取 InterfaceDef 对象引用，这里 conRef 是实现 Control 接口的分布式对象的引用。

(2) 以操作的名字 openAccount 为参数调用 lookup()方法，来检索所需要的 OperationDef 对象。

(3) 通过调用 OperationDef 对象中的 describe()方法来获取 OperationDescription 对象。这里，OperationDescription 是在 OperationDef 接口中定义的 struct 类型，用于保存与操作有关的信息，如操作名、参数名以及参数类型等。这样，就可以获取 openAccount()操作的信息。

6.3.4 仓库 ID

所谓仓库 ID 是指用于识别存放在接口仓库中的 IDL 信息的字符串，它是一个全局的名字，主要用于识别接口仓库中的接口和操作等。仓库 ID 可以在接口定义中由程序设计者

明确地给出，如果没有指定的话，将采用如下的表示形式：

IDL：模块名/接口名：1.0

其中，IDL 表示所采用的仓库 ID 的格式；“模块名/接口名”部分是模块与接口等的完整路径(作用域名)；1.0 是版本号，默认为 1.0。这三部分是由冒号(：)分隔的。

例如，模块 Bank 下的 Control 接口的仓库 ID 的格式如下：

```
IDL:Bank/Control:1.0
```

除了接口之外，被保存在接口仓库中的对象基本上都被赋予特定的仓库 ID。

6.4 动态启动接口 DII

前面已经介绍了利用 Stub 来进行 CORBA 客户端程序设计的方法，在本节将简单介绍利用动态启动接口 DII 来设计 CORBA 客户端程序的方法。

利用 DII 进行程序设计时，不需要利用 IDL 接口经编译后所生成的 Stub 类，而是在程序运行过程中通过动态地生成请求信息来实现分布式对象的调用。

尽管利用 Stub 类进行程序设计时并不需要理解 DII 的功能，但了解 DII 的处理过程将有益于对 CORBA 系统的理解。

6.4.1 DII 程序设计过程

在利用 DII 设计 CORBA 客户端程序时，主要是由如下几部分组成的：

① ORB 的初始化。

② 分布式对象引用的获取。

③ Request 对象的生成。

④ 参数的设置。

⑤ 方法的启动。

⑥ 异常的检查。

⑦ 返回值的取出。

ORB 的初始化与分布式对象引用的获取，在前面已经介绍过了，在此不再重复。下面将从 Request 对象的生成开始逐一进行介绍。

6.4.2 Request 对象

Request 对象用于隐藏在进行对象调用时所必需的所有信息。其中包括对象引用、被调用对象的操作名(或属性名)、保存参数的 NVList 对象、保存返回值的 NamedValue 对象、异常信息以及执行对象调用过程的方法等。

1. Request 接口

Request 接口的 Java 映射结果如下：

```
package org.omg.CORBA;
```

```
public abstract class Request
{
    public abstract Object target();
    public abstract String operation();
    public abstract NVList arguments();
    public abstract NamedValue result();
    public abstract Environment env();
    public abstract ExceptionList exceptions();
    public abstract ContextList contexts();
    public abstract Context ctx();
    public abstract void ctx(Context c);
    public abstract Any add_in_arg();
    public abstract Any add_named_in_arg(String name);
    public abstract Any add_inout_arg();
    public abstract Any add_named_inout_arg(String name);
    public abstract Any add_out_arg();
    public abstract Any add_named_out_arg(String name);
    public abstract void set_return_type(TypeCode tc);
    public abstract Any return_value();
    public abstract void invoke();
    public abstract void send_oneway();
    public abstract void send_deferred();
    public abstract void get_response()throws org.omg.CORBA.WrongTransaction;
    public abstract boolean poll_response();
}
```

说明：在 Request 接口中，定义了为请求对象设置信息的方法，如 add_out_arg()和 set_return_type()等；定义了取出调用结果的方法，如 result()和 env()等；同时还定义了执行对象调用过程的方法，如 invoke()、send_oneway()和 send_deferred()等。

2. Request 对象的生成

利用 DII 的第一步是生成 Request 对象，该对象用于实现 Request 接口，而 Request 接口中提供了启动方法的操作。

为了生成 Request 对象需要利用 org.omg.CORBA.Object 中的_request()方法或者_create_request()方法。同时，在 Request 对象生成之前，需要先获取分布式对象引用，这是因为 Request 对象的生成是利用分布式对象引用来调用这两个方法实现的。这两个方法的定义如下所示：

```
package org.omg.CORBA;
public interface Object
{
    Request _request(String s);
    Request _create_request(Context ctx,String operation,
                            NVList arg_list,NamedValue result);
    Request _create_request(Context ctx,String operation,
                            NVList arg_list,NamedValue result,
                            ExceptionList exclist,ContextList ctxlist);
    ⋮
}
```

说明：Object 接口中的 Request 对象生成方法，可以分为两种：一种是_request()方法，该方法以方法名或属性名为参数来生成 Request 对象，而参数和返回值等信息则在生成了 Request 对象以后，通过调用 Request 对象中的方法来设置；第二种是_create_request()方法，这两个方法(参数不同)在调用之前应准备好参数，然后生成 Request 对象。

下面简单介绍一下_request()和_create_request()方法的使用过程及其参数设置方法。

(1) _request()方法的使用及其参数设置。

在利用_request()方法来生成 Request 对象时，除了 target(当前对象引用)和 operation(操作名)之外，其余的数据都没有设置。这些没有设置的数据，在方法启动之前，必须利用某种方法来设置。一般来讲，可以利用两种方法来设置参数：一是利用 Request 对象的 add_in_arg()等方法来设置参数，二是利用 Request 对象的 arguments()方法和 NVList 对象来设置参数。

① 利用 add_in_arg()等方法来设置参数。

下面的程序段给出了这种参数设置过程。

```
//生成 Request 对象
Request conReq = conRef._request("openAccount");
//向 Request 对象中设置参数
conReq.add_in_arg().insert_string(acc_var);
conReq.add_in_arg().insert_string(pwd_var);
//向 Request 对象中设置返回值的类型
conReq.set_return_type(Bank.AccountHelper.type());
```

说明：这里的 conRef 是 Control 对象的引用。在设置参数时，首先利用 add_in_arg()等方法生成存放参数的 Any 对象，然后利用 Any 对象中的 insert_string()等方法向其中设置参数。当然，如果参数的类型是用户定义类型的话，就需要利用相应类型的 Helper 类中的 insert()方法来设置参数。参数的设置也包括返回值类型的设置。所有参数都设置完了以后，就可以动态启动调用请求了。

② 利用 arguments()方法与 NVList 对象来设置参数。

下面的程序段给出了这种参数设置过程。

```
//生成 Request 对象
Request conReq = conRef._request("openAccount");
//向 Request 对象中设置参数
(conReq.arguments().add(ARG_IN.value)).value().insert_string(acc_var);
(conReq.arguments().add(ARG_IN.value)).value().insert_string(pwd_var);
//向 Request 对象中设置返回值的类型
(conReq.result().value()).type(Bank.AccountHelper.type());
```

说明：在本程序段中，conReq.arguments()方法从 Request 对象中返回 NVList 对象。NVList 是用于保存名字及其值的对应关系的列表。只要理解了 NVList 和 NamedValue 两个类的定义，就不难理解本程序段的意义。

(conReq.arguments().add(ARG_IN.value)).value()用于生成保存最初参数的 NamedValue 数据的 Any 对象，然后，就可以利用 insert_string()等方法向该 Any 对象中插入数据。

由于返回值是通过 result() 方法返回的 NamedValue 对象，因此，在设置返回值类型

时，只要为其设置 Any 对象的数据类型即可。

(2) _create_request()方法的使用及其参数设置。

在利用_create_request()方法来生成 Request 对象之前，必须准备好所需要的参数。下面的程序段给出了这种参数设置过程。

```
//NVList 对象的参数设置
NVList arguments = orb.create_list(2);
(arguments.add(ARG_IN.value)).value().insert_string(acc_var);
(arguments.add(ARG_IN.value)).value().insert_string(pwd_var);
//返回值的数据类型设定
Any any_var = orb.create_any();
any_var.type(Bank.AccountHelper.type());
NamedValue result = orb.create_named_value("",any_var,0);
Context ctx_var = orb.get_default_context();
//conRef 是实现 Control 接口的对象的引用
Request conReq = conRef._create_request(ctx_var,"openAccount",
                                        arguments,result);
```

说明：在调用_create_request()方法生成 Request 对象之前，需要预先准备好这样几个参数：保存操作参数的 NVList 对象(arguments)、保存返回值的 NamedValue 对象以及必须要指定的 Context 对象(用于传递 Context 数据)。以这些对象为参数，就可以利用_create_request()方法来生成 Request 对象。在此基础上，就可以动态启动调用请求了。需要注意的是，调用_create_request()方法需要利用分布式对象的引用，这里的 conRef 就是分布式对象引用。

6.4.3 动态启动调用请求

生成了 Request 对象并设置了参数之后，就可以将请求信息发送给服务器，以便调用分布式方法。在 DII 中，根据获取返回值方式的不同，可以采用如下几种方式向服务器发送调用请求。

1. 同步方式

同步方式是利用 Request 对象中的 invoke()方法来发送调用请求，然后处于等待状态，直到获取返回结果为止。

2. oneway 方式

oneway 方式是利用 send_oneway()方法来发送调用请求，在将请求信息发送出去之后并不等待返回结果，而是直接将控制返回给调用者。因此，从 send_oneway()方法返回之后，并不表示要调用的方法已经执行结束，也不能保证要调用的方法是否结束。

即使在 IDL 定义中没有被指定为 oneway 的方法，也可以利用 send_oneway()方法来发送调用请求。

在利用 oneway 方式时，方法的参数属性不能被设置为 out 和 inout，其返回值类型也需要指定为 void。

3. 延迟同步方式

前两种请求信息发送方式可以由 IDL 指定，也就是在利用 IDL 来定义操作时，如果没有加 oneway 关键字，则将采用同步调用方式，否则将采用 oneway 调用方式。直接利用 Stub 类来设计 CORBA 客户端时支持这两种方式，而延迟同步方式只能在利用 DII 进行程序设计时使用。

延迟同步方式是利用 send_deferred()方法来发送请求信息的，同时，通过利用 get_response()方法来等待返回值。下面的程序段给出了这两个方法的具体用法。

```
//conReq 是已设定数据的 Request 对象
conReq.send_deferred();
⋮
//利用 get_response()方法等待返回值
conReq.get_response();
```

延迟同步方式可以通过检测返回值的状态来进行相应的处理，为此，提供了 poll_response()方法，下面的程序段给出了该方法的具体用法。

```
//conReq 是已设定数据的 Request 对象
conReq.send_deferred();
//利用 poll_response()方法来检测返回值
while(conReq.poll_response() == false)
{
    //执行别的处理
    ⋮
}
//利用 get_response()取出返回值
conReq.get_response();
```

说明：poll_response()方法在检查了返回值的状况之后立即返回，如果 poll_response()方法的返回值为 false，则说明返回值还未到达，这时可以进行别的处理；如果其返回值为 true，则说明返回值已经到达，此时，可以利用 get_response()方法来获取返回值。实际上，这里所说的获取返回值只是将返回值保存在 Request 对象中，真正取出返回值还需要利用下面的方法。

6.4.4 返回值的取出

分布式对象调用结束之后，就可以从 Request 对象中取出处理结果。为了取出处理结果，可以利用 Request 对象中的 result()方法、arguments()方法和 env()等方法。result()方法返回 NamedValue 类型的返回值，arguments()方法返回 NVList 类型的参数值（如果参数的属性是 out 或 inout），env()方法用于返回 Environment 类型的用户定义的异常信息。由于 NamedValue 类型与 NVList 类型的定义在前面已经介绍过，这里只给出 Environment 类型的定义。

```
package org.omg.CORBA;
public abstract class Environment
{
```

```
    public abstract void exception(java.lang.Exception except);
    public abstract java.lang.Exception exception();
    public abstract void clear();
}
```

说明：通过 env()方法获取了 Environment 对象之后，就可以利用该对象中的 exception()方法来获取与用户定义类型相对应的异常对象。

下面的程序段给出了调用请求发出之后返回值的取出过程。

```
//以同步方式发送调用请求
conReq.invoke();
//返回值的取出
Bank.Account acc_var = Bank.AccountHelper.extract(conReq.result().value());
```

这里，由于 Account 是用户定义的接口类型，因此利用接口的 Helper 类中的 extract()方法从 Any 对象中取出返回值中。

前面已经介绍过，在利用 DII 进行客户端程序设计时，是不需要使用 Stub 类的，但实际上 Stub 类本身就是使用 DII 来实现的。仔细地分析 Stub 类的构成对于理解 DII 的应用过程是有益处的，同时，学习了本节的内容之后，对 Stub 类的理解也就比较容易了。

6.5 动态骨架接口 DSI

前面已经介绍了利用 Skeleton 来进行 CORBA 服务器程序设计的方法，在本节将简单介绍利用动态骨架接口 DSI 来设计 CORBA 服务器程序的方法。

利用 DSI 进行程序设计时，不需要利用 IDL 接口经编译之后所生成的 Skeleton 类，而是在程序运行过程中动态地接收来自客户端的请求，并将请求分发给相应接口中的相应操作。

尽管利用 Skeleton 类进行程序设计时并不需要理解 DSI 的功能，但了解 DSI 的处理过程将有益于对 CORBA 系统的理解。

6.5.1 DynamicImplementation 类

在利用 DSI 实现 Servant(服务对象)类时，必须要继承 DynamicImplementation 类，下面的程序段给出了该类的定义。

```
package org.omg.PortableServer
public abstract class org.omg.CORBA.DynamicImplementation extends Servant
{
    public abstract void invoke(org.omg.CORBA.ServerRequest request);
}
```

说明：在 DynamicImplementation 类中定义了一个抽象方法 invoke()，在客户端调用服务器的方法或属性时，invoke()方法将被调用。在 Servant 类中通过实现该方法，将实现对方法和属性的调用过程。

invoke()方法有一个 ServerRequest 类型的参数，该参数封装了与对象调用有关的所有信息，从此参数中可以获取调用一个方法所需要的方法名和参数值等信息。同时，在方法执

行结束时，执行结果也被保存在 ServerRequest 对象中。

在利用 DSI 来实现服务器类时，必须要给出作为 DynamicImplementation 父类的 Servant 类的_all_interfaces()方法的定义。_all_interfaces()方法用于返回包括父接口在内的服务器对象的所有接口的仓库 ID，该方法主要是在生成对象引用时在 ORB 内部进行调用。

6.5.2 ServerRequest 接口

ServerRequest 接口被映射为 Java 语言的抽象类，下面的程序段给出了该类的定义。

```
package org.omg.CORBA;
public abstract class ServerRequest
{
    public abstract String op_name();
    public abstract String operation();
    public abstract Context ctx();
    public abstract void params(NVList);
    public abstract void arguments(NVList);
    public abstract void result(Any);
    public abstract void set_result(Any);
    public abstract void except(Any);
    public abstract void set_exception(Any);
    ⋮
}
```

说明：

(1) ServerRequest 对象用于封装被调用对象的方法名(或属性名)、Context 对象、参数信息、返回值信息以及异常信息等。

(2) operation()方法用于获取方法名或属性名。

(3) arguments()方法用于获取输入参数或设置输出参数。

(4) set_result()方法用于设置返回值。

(5) set_exception()方法用于设置异常信息。

前面已经介绍过，在利用 DSI 进行服务器程序设计时，是不需要使用 Skeleton 类的，但实际上 Skeleton 类本身就是使用 DSI 来实现的。仔细地分析 Skeleton 类的构成对于理解 DSI 的应用过程是有益处的，同时，学习了本节的内容之后，对 Skeleton 类的理解也就比较容易了。

第7章 CORBA 实例

在前面几章中比较详细地介绍了分布式对象的基本概念、IDL 语言以及基于 CORBA 的分布式对象系统的构建细节，从原理上完整地给出了 CORBA 客户端和服务器的设计过程，并给出了大量的示例。由于不同的 CORBA 系统其程序设计过程也是有差异的，本章将通过几个具有应用价值的实例，利用环境易于构建的 Java IDL 来说明 CORBA 应用系统的构建过程，主要包括运行环境配置、IDL 接口定义、IDL 到 Java 语言映射、服务器程序设计、客户端程序设计、Java 语言编译、Java IDL 命名服务的启动以及服务器程序和客户端程序的运行等。本章介绍的程序都是经过严格测试的，在给定环境中能够实际运行的。感兴趣的读者可以按照本章的内容来构建环境和设计 CORBA 程序，开始的时候可以按照本章所给定的步骤来调试本章的程序，然后就可以设计和实现自己的 CORBA 应用程序了。

7.1 Java IDL 及其应用系统开发过程

Java IDL 是由 Java 2 提供的满足 CORBA 规范的分布式对象技术，它提供了从 IDL 到 Java 语言的映射。Java IDL 除了支持用 Java 语言写的对象之间的交互之外，同时也支持用其他语言写的对象之间的相互交互，这是由于它是基于 CORBA 技术的。

基于 Java IDL 的分布式对象系统开发过程与一般的 CORBA 应用系统的开发过程是一样的，主要是由以下几部分组成的。

1. IDL 接口定义

将希望利用分布式对象实现的方法和属性等利用 IDL 语言进行描述，只有在 IDL 接口中描述的方法才能作为分布式方法进行实现。

2. IDL 到 Java 语言的映射

利用 IDL 编译器将 IDL 接口定义映射为 Java 语言，作为映射结果的 Stub 和 Skeleton 等类将被用于实现客户端程序和服务器程序。

3. 服务器程序实现

服务器程序主要是由两部分组成的：一是分布式对象实现，二是服务器进程程序。分

布式对象实现可以利用 Skeleton 继承方式来实现，服务器进程的主要任务包括分布式对象的生成并将其登录到 ORB 中、分布式对象引用的生成并将其登录到命名服务中以及等待来自客户端的调用请求等几部分。

4. 客户端程序实现

客户端程序主要是利用 Stub 类来实现的，通过 Java IDL 提供的命名服务来获取分布式对象的引用，通过该引用来调用分布式方法，进而实现所需要的功能。

5. 系统运行

上述工作完成之后，客户端与服务器的程序设计也就完成了，这样，就可以启动命名服务和启动服务器程序，并在此基础上运行客户端程序。

7.2 环境配置

CORBA 程序设计的第一步是进行环境配置，只有正确地配置了运行环境，才能顺利地实现客户端程序和服务器程序。由于 Java IDL 是 Java 2 的一个组成部分，因此，对 Java IDL 的环境配置与 Java 2 的环境配置是一样的，也是比较容易配置的。

本章所给出程序都是在下列环境下调试完成的。

1. 操作系统与语言环境

操作系统是 Windows 2000，所采用的 Java 2 版本是 J2SE SDK 1.4.2。

2. 环境变量设置

环境变量的设置方法与 Java 2 运行环境的设置是一样的，这里假设 Java 2 被安装在 c:\j2sdk1.4.2_06 目录下。需要设置的环境变量主要包括 classpath、java_home 和 path 等，这些环境变量的具体设置内容如下：

```
classpath = ".;c:\j2sdk1.4.2_06\lib\tools.jar;c:\j2sdk1.4.2_06\lib\dt.jar;
  c:\j2sdk1.4.2_06\jre\lib;…"
java_home = "c:\j2sdk1.4.2_06"
path = "c:\j2sdk1.4.2_06\bin;…"
```

为了检验环境配置的是否成功，可以编写一个简单的 Java 语言程序，并通过编译和运行等步骤来判断 Java 语言环境是否正常建立起来了。

7.3 CORBA 实例 1：一般属性和操作的定义与使用

本节的例题主要用于介绍 CORBA 对一般属性和操作（方法）的处理过程，通过对该例题的学习，能够使读者很好地掌握 CORBA 的基本程序设计与实现过程。

7.3.1 问题描述与 IDL 接口定义

本实例用于管理用户的银行账户信息，尽管功能简单，但包含了存款、取款以及余额查询等功能。设计本程序的主要目的是使读者能够了解 IDL 的基本用法，进一步了解属性和操作(方法)的基本定义方式。其 IDL 接口定义如下：

```
//Bank.idl
module Bank
{
    interface Account
    {
        readonly attribute unsigned long balance;
        unsigned long Deposit(in unsigned long amount);
        unsigned long Withdraw(in unsigned long amount);
    };
};
```

说明：

(1) module 关键字用于定义模块，当进行映射时，作为模块名的 Bank 将被映射为 Java 语言的包(package)名。

(2) interface 关键字用于定义接口，作为分布式对象实现的操作和属性往往都是在接口中定义的，同时，一个 interface 定义一般对应一个分布式对象实现。在进行语言映射时，interface 定义被映射为 Java 语言的接口、Stub 和 Skeleton 等类。

(3) balance 属性用于对余额进行管理，由于被定义为只读属性，因此，只能获取余额，而不能设置余额。

(4) Deposit( )用于存款操作，参数表示存款的数量，返回值为存款后余额。

(5) Withdraw( )用于取款操作，参数表示取款的数量，返回值为取款后余额。

根据所需要的功能定义了 IDL 接口之后，还需要将 IDL 接口映射为 Java 语言，这样，才能利用 Java 语言来实现 CORBA 客户端和服务器。

7.3.2 IDL 到 Java 语言的映射

所谓“IDL 到 Java 语言的映射”，是指通过编译 IDL 接口定义，将其映射为 Java 语言的类，以便能够用于基于 Java 的 CORBA 客户端和服务器设计。

假如上述 IDL 接口定义被保存在名为 Bank.idl 的文件中，该文件所在目录为 C:\CORBA，则其映射步骤如下：

```
C:\CORBA>idlj -fall Bank.idl
C:\CORBA>
```

其中，idlj 为编译器的可执行文件名，其参数-f<side>：<side>的可选值为 client、server 和 all 等，表示要生成哪一端使用的程序。如果选择 all，则表示既生成客户端使用的程序又生成服务器端使用的程序，其默认值为-fclient。

经过上述编译之后，将生成如表 7.1 所示的 Java 映射结果。需要注意的是，该映射结

果是基于 POA 生成的。

表 7.1 IDL 到 Java 语言的映射结果

源 文 件	说 明
AccountPOA. java	用于服务器设计的 skeleton 类 extends org. omg. PortableServer. Servant implements Bank. AccountOperations, org. omg. CORBA. portable. InvokeHandler
_AccountStub. java	用于客户端设计的 stub 类 extends org. omg. CORBA. portable. ObjectImpl implements Bank. Account
Account. java	Java 语言的接口 extends AccountOperations, org. omg. CORBA. Object, org. omg. CORBA. portable. IDLEntity
AccountOperations. java	不继承任何接口的 Java 语言的接口
AccountHelper. java	Helper 类
AccountHolder. java	Holder 类

下面对表 7.1 中的映射结果进行简单的说明。

(1) AccountPOA. java 是 Skeleton 类所在的文件，该类主要用于设计服务器程序。这里是基于 POA 生成的 Skeleton 类，在其所继承的类以及所要实现的接口方面，与前面介绍的基于 BOA 的情况是不同的。

(2) _AccountStub. java 是 Stub 类所在的文件，该类主要用于设计客户端程序。由该类所继承的类以及所需要实现的接口来看，与前面介绍的情况是相同的。

(3) Account. java 和 AccountOperations. java 是与 IDL 对应的 Java 语言的接口所在的文件。这两个接口的主要区别就在于 Account 接口需要继承 org. omg. CORBA. Object 接口，而 AccountOperations 接口不需要继承该接口。

(4) AccountHelper. java 是 Account 接口的 Helper 类所在的文件，主要提供 narrow( )等方法。

(5) AccountHolder. java 是 Account 接口的 Holder 类所在的文件，主要用于实现 out 和 inout 属性的参数传递。

7.3.3 服务器端的 Java 语言程序设计

在上述接口定义及其映射结果的基础上，就可以实现 CORBA 服务器程序和客户端程序了。本节将给出服务器程序的实现过程，包括分布式对象实现类和服务器进程类两部分。该服务器程序被保存在 BankCorbaServer. java 文件中。

```
//BankCorbaServer.java
import Bank.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties;
```

```
//分布式对象实现类
class AccountImpl extends AccountPOA
{
    private ORB orb;
    private int balanceVar;
    public void setORB(ORB orb_val)
    {
        orb = orb_val;  //保存 ORB 对象引用
    }
    public AccountImpl( )
    {
        balanceVar = 0;
    }
    public int Deposit(int amount)
    {
        balanceVar += amount;
        return balanceVar;
    }
    public Withdraw(int amount)
    {
        balanceVar -= amount;
        return balanceVar;
    }
    public int balance( )
    {
        return balanceVar;
    }
}
//服务器进程类
public class BankCorbaServer
{
    public static void main(String args[])
    {
        try
        {
            // ORB 的初始化
            ORB orb = ORB.init(args,null);
            // 获取 RootPOA 的引用,并使 POAManager 变为活动状态
            POA rootpoa =
                    POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager().activate();
            // 生成 Servant,并将 ORB 对象保存起来
            AccountImpl accountImpl = new AccountImpl();
            accountImpl.setORB(orb);
            //由 Servant 来获取 Account 对象的引用
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(accountImpl);
            Account href = AccountHelper.narrow(ref);
            // 获取命名服务对象的引用
            org.omg.CORBA.Object objRef =
                                orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            // 将 Account 对象的引用登录到命名服务中
            String name = "Account";
            NameComponent path[] = ncRef.to_name( name );
```

```
            ncRef.rebind(path,href);
            System.out.println("BankCorbaServer 已经启动了!");
            // 等待来自客户端的调用请求
            orb.run();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

说明：由于服务器程序是由两部分组成的，下面分别进行说明。

(1) 分布式对象实现部分，即 AccountImpl 类。该类是通过继承 Skeleton 类 AccountPOA 而生成的。在该类中，对 IDL 中的只读属性 balance 是利用 balance( )来实现的，在该方法中只是返回存款余额；Deposit( )方法和 Withdraw( )方法分别用于实现在 IDL 接口中定义的两个操作。

(2) 服务器进程部分，即 BankCorbaServer 类。该类的定义同一般的 Java 应用程序相似，主要是由如下一些基本内容组成的。

① 导入必要的包(package)、定义服务器进程类、定义 main( )方法以及异常处理等。这里，类名可以是任意的标识符，对异常的处理过程也是通过 try-catch 来进行的。

② ORB 的初始化，获取 ORB 对象的引用。服务器需要一个本地的 ORB 对象来执行所有和 IIOP 协议有关的任务，每个服务器实例化一个 ORB 对象并注册服务对象，以便 ORB 收到调用请求时能够找到相应的服务。ORB 的初始化是利用下述语句来实现的：

```
ORB orb = ORB.init(args,null);
```

在 init( )方法中使用命令行参数 args，以利于在运行时确定其特性。

③ 本服务器程序采用 POA 技术，这样，首先需要获取根 POA 对象的引用，并激活 POA 管理器。根据 POA 对象引用的获取方式与命名服务的获取方式相同，是利用下述语句实现的：

```
orb.resolve_initial_references("RootPOA");
```

其中，RootPOA 是用于获取根 POA 的参数。

④ 生成服务对象(Servant)，并据此生成分布式对象引用。服务对象的生成同本地对象的生成是一样的，都是利用 new 运算符来实现的。下面的语句用于生成服务对象：

```
AccountImpl accountImpl = new AccountImpl();
```

下面的语句用于从服务对象来生成分布式对象引用：

```
org.omg.CORBA.Object ref = rootpoa.servant_to_reference(accountImpl);
```

⑤ 获取命名服务对象引用，并将 Account 分布式对象引用登录到命名服务中。命名服务对象引用的获取方式以前已经介绍过，通过调用命名服务对象中的方法将分布式对象引用登录到命名服务中。

⑥ 等待来自客户端的调用请求。下面的语句就是用于实现此功能的：

```
orb.run();
```

⑦ 在本服务器程序中出现了多个 narrow( )方法，此方法的意义在前几章中已经介绍过，在此不再赘述。

7.3.4 客户端的 Java 语言程序设计

本节将给出客户端程序的实现过程，该客户端程序被保存在 BankCorbaClient.java 文件中。

```
//BankCorbaClient.java
import Bank.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class BankCorbaClient
{
    static Account AccountImpl;
    public static void main(String args[])
    {
        try
        {
            // ORB 的初始化
            ORB orb = ORB.init(args,null);
            // 获取命名服务对象的引用
            org.omg.CORBA.Object objRef =
                                orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            // 从命名服务中获取 Account 对象的引用
            String name = "Account";
            AccountImpl = AccountHelper.narrow(ncRef.resolve_str(name));
            // 调用 Deposit()方法
            AccountImpl.Deposit(100);
            //调用实现只读属性的方法 balance( )
            System.out.println("Balance = " + AccountImpl.balance( ));
            //调用 Withdraw( )方法
            int bm = AccountImpl.Withdraw(30);
            System.out.println("Balance = " + bm);
            bm = AccountImpl.Withdraw(20);
            bm = AccountImpl.balance( );
            System.out.println("Balance = " + bm);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

说明：CORBA 客户端程序主要是利分布式对象来实现相应的功能，当然，在客户端程序中也可以调用本地对象。本客户端程序主要是由如下一些基本内容组成的。

（1）导入所需要的包、定义客户端程序类和 main( )函数。客户端程序的类名可以是任意的标识符。

（2）在 main( )函数中，首先获取 ORB 对象的引用，然后获取命名服务对象，并根据在服务器程序中将 Account 对象引用登录到命名服务时所使用的名字，从命名服务中获取 Account 对象应用。

（3）一旦获取了分布式对象应用，就可以像本地对象一样来调用分布式对象中的方法。比如，下面的语句用于调用 AccountImpl 对象中的 Deposit( )方法，其入口参数是 100：

```
AccountImpl.Deposit(100);
```

7.3.5 Java 类的编译

上面定义的服务器程序、客户端程序以及 IDL 映射之后所生成的 Stub 和 Skeleton 等程序都是源程序，是不能直接运行的，在运行之前必须要进行编译。由前述可知，当前工作目录是 CORBA，也就是 IDL 接口定义、服务器程序和客户端程序都被保存在 CORBA 目录下，这样，由于 IDL 接口定义中的模块名是 Bank，因此映射后的结果将被保存在 CORBA\Bank\目录下，这样，可采用下述命令对源程序进行编译：

```
C:\CORBA>javac *.java Bank\*.java
C:\CORBA>
```

7.3.6 启动 orbd

orbd 是 Object Request Broker Daemon 的简称，是 ORB 的一个程序。借助于 orbd，客户端可以利用命名服务来获得对象引用，进而可以利用分布式对象。

orbd 的启动命令如下：

```
C:\>start orbd -ORBInitialPort 1050
```

说明：作为命令行参数，需要指定端口号(port)。

需要注意的是，如果 orbd 正常启动了，则画面上不显示任何内容。

7.3.7 服务器端程序的执行

orbd 正常启动了之后，就可以启动服务器程序了。另开一个窗口后，在 DOS 命令行上输入如下命令：

```
C:\CORBA>java BankCorbaServer -ORBInitialPort 1050 -ORBInitialHost localhost
```

将显示如下信息：

```
BankCorbaServer 已经启动了!
```

说明：作为命令行参数，-ORBInitialPort 1050 -ORBInitialHost localhost 用于指定正在运行 ORB 的 orbd 程序的服务器的 port 号和 host 名。由于是在 localhost 上运行的，所以如上述那样指定即可。

需要注意的是，CORBA 服务器程序启动以后，就处于接收调用请求的状态。

7.3.8 客户端程序的执行

CORBA 服务器程序启动了之后，就可以运行客户端程序了。另开一个窗口后，在 DOS 命令行上输入如下命令：

```
C:\CORBA>java BankCorbaClient -ORBInitialPort 1050 -ORBInitialHost localhost
```

屏幕上将显示如下执行结果：

```
Balance = 100
Balance = 70
Balance = 50
```

如果按照上述命令格式再执行一次客户端程序，则将显示如下执行结果：

```
Balance = 150
Balance = 120
Balance = 100
```

从运行结果可以看出，一旦分布式对象创建了之后，就同本地对象一样，是可以进行多次调用的，同时其运行结果也可以保持连续性。

至此，整个 CORBA 实例 1 应用程序的开发和运行过程就结束了。

7.4 CORBA 实例 2：本地方法与 Holder 类的使用

本节的例题主要用于介绍在 CORBA 程序设计过程中经常会遇到的 Holder 类的用法，包括基本类型的 Holder 类和用户定义类型的 Holder 类，进一步了解 Holder 类在参数传递过程中的作用。同时，说明本地方法的定义与使用。通过对该例题的学习，能够使读者更好地掌握 CORBA 的复杂程序设计与实现过程。

7.4.1 问题描述与 IDL 接口定义

本例题主要考察 CORBA 中 Holder 类的用法，对 Holder 类在基本数据类型以及用户定义类型的参数传递过程中的作用进行介绍。同时，学习在分布式对象实现中定义本地方法的情况。下面给出其 IDL 接口定义。

```
//Test.idl
module Test
{
    interface Istruct
    {
        struct sData
        {
            string str;
            long len;
        };
        void getValue( out sData value);
```

```
        void setValue( in sData value);
        void setString(in string str);
        void getString(out string str);
        oneway void shutdown();
    };
};
```

说明：

(1) sData 是结构类型，是用户定义类型之一。如前所述，每个用户定义类型在 IDL 映射时都将被映射为相应的 Holder 类和 Helper 类。

(2) setValue( )方法用于设置结构类型变量，也就是将其入口参数的结构值保存在分布式对象中，getValue( )方法用于取出结构类型的变量值，也就是将保存在分布式对象中的结构变量的值取出来，以便利用。这里需要注意的是，由于 getValue( )方法的参数属性是 out，因此，是通过参数将对象中的结构值带回客户端，而不是通过返回值带回。

(3) setString( )方法用于设置字符串类型变量，也就是将其入口参数的字符串保存在分布式对象中，getString( )方法用于取出字符串类型的变量值，也就是将保存在分布式对象中的字符串变量的值取出来，以便利用。这里需要注意的是，由于 getString( )方法的参数属性是 out，因此，是通过参数将对象中的字符串带回客户端，而不是通过返回值带回。由于 string 是 IDL 中的基本数据类型，因此，string 类型的 Holder 类是预先定义好的，而不是通过映射生成的。

(4) shutdown( )方法通过调用 ORB 对象中的 shutdown( )方法来关闭 ORB 的运行，从而停止 CORBA 服务器程序的运行。

7.4.2 IDL 到 Java 语言的映射

假如上述 IDL 接口定义被保存在名为 Test.idl 的文件中，该文件所在目录为 C:\CORBA1，则其映射步骤如下：

```
C:\CORBA1>idlj -fall Test.idl
C:\CORBA1>
```

经过上述编译之后，将生成如表 7.2 所示的 Java 映射结果。需要注意的是，该映射结果是基于 POA 生成的。

表 7.2 IDL 到 Java 语言的映射结果

源 文 件	说 明
IstructPOA.java	用于服务器设计的 skeleton 类(包名为 Test) extends org.omg.PortableServer.Servant implements Test.IstructOperations, org.omg.CORBA.portable.InvokeHandler
_IstructStub.java	用于客户端设计的 stub 类(包名为 Test) extends org.omg.CORBA.portable.ObjectImpl implements Test.Istruct
Istruct.java	Java 语言的接口(包名为 Test) extends IstructOperations, org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity

续表

源 文 件	说 明
IstructOperations. java	不继承任何接口的 Java 语言的接口(包名为 Test)
IstructHelper. java	Helper 类(包名为 Test)
IstructHolder. java	Holder 类(包名为 Test) implements org. omg. CORBA. portable. Streamable
sData. java	sData 结构类型所映射的 Java 语言的类(包名为 Test. IstructPackage) implements org. omg. CORBA. portable. IDLEntity
sDataHolder. java	sData 结构类型所映射的 Holder 类(包名为 Test. IstructPackage) implements org. omg. CORBA. portable. Streamable
sDataHelper. java	sData 结构类型所映射的 Helper 类(包名为 Test. IstructPackage)

下面对表 7.2 的映射结果进行简单的说明。

(1) IstructPOA. java 是 Istruct 接口所对应的 Skeleton 类。

(2) _IstructStub. java 是 Istruct 接口所对应的 Stub 类。

(3) Istruct. java 和 IstructOperations. java 是 Istruct 接口所对应的 Java 语言接口。

(4) IstructHelper. java 和 IstructHolder. java 分别是 Istruct 接口所对应的 Helper 类和 Holder 类。

(5) sData. java 是 sData 结构类型经映射之后所生成的 Java 语言的类。

(6) sDataHolder. java 和 sDataHelper. java 分别是 sData 结构类型所对应的 Holder 类和 Helper 类。

7.4.3 服务器端的 Java 语言程序设计

下面是 CORBA 服务器程序,该程序被保存在 TestCorbaServer. java 文件中。

```
//TestCorbaServer.java
import Test.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties;

//分布式对象实现类
class IstructImpl extends IstructPOA
{
    private ORB orb;
    //声明两个 Holder 类成员
    Test.IstructPackage.sDataHolder sHolderValue;
    StringHolder strHolder;
    //setORB( )是本地方法
    public void setORB(ORB orb_val)
    {
        //保存 ORB 对象
```

```
        orb = orb_val;
    }
    //定义构造函数
    public IstructImpl( )
    {
        //创建两个 Holder 类对象
        sHolderValue = new Test.IstructPackage.sDataHolder(null);
        strHolder = new StringHolder(null);
    }
    //定义与 IDL 接口对应的方法
    public void setValue(Test.IstructPackage.sData sH)
    {
        sHolderValue.value = sH;
    }
    //利用 Holder 类参数向客户端传递用户定义类型数据
    public void getValue(Test.IstructPackage.sDataHolder sH)
    {
        sH.value = sHolderValue.value;
    }
    public void setString(String str)
    {
        strHolder.value = str;
    }
    //利用 Holder 类参数向客户端传递基本类型数据
    public void getString(StringHolder str)
    {
        str.value = strHolder.value;
    }
    public void shutdown( )
    {
        //关闭 ORB 对象
        orb.shutdown(false);
    }
}

//服务器进程类
public class TestCorbaServer
{
    public static void main(String args[])
    {
        try
        {
            //ORB 对象的获取
            ORB orb = ORB.init(args,null);
            //rootPOA 对象的获取与 POA 管理器的激活
            POA rootpoa =
                POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager().activate();
            //创建分布式对象
            IstructImpl IstructImpl = new IstructImpl();
            IstructImpl.setORB(orb);
```

```
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(IstructImpl);
            Istruct href = IstructHelper.narrow(ref);
            //获取命名服务对象
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            String name = "Istruct";
            NameComponent path[] = ncRef.to_name( name );
            ncRef.rebind(path,href);
            System.out.println("TestCorbaServer starting...");
            //等待接收来自客户端的调用请求
            orb.run();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

说明：

(1) 由分布式对象的实现类 IstructImpl 的定义可知，IDL 接口中 in 属性的用户定义类型的参数是通过其被映射的 Java 语言类来定义的，而 out 属性的用户定义类型的参数则是通过其被映射的 Java 语言的 Holder 类来定义的，同样，in 属性的 string 基本类型参数是通过 java.lang.String 来定义的，而 out 属性的 string 基本类型参数则是通过 org.omg.CORBA.StringHolder 来定义的。

(2) 对 Holder 类的数据存取过程是与 Holder 类的定义密不可分的，为了正确运用 Holder 类进行程序设计，应该掌握 Holder 类的定义形式。

(3) 在分布式对象实现类中定义的 setORB( )方法是本地方法，不能作为分布式方法被客户端所调用，它只能由服务器对象所调用，这是由于在 IDL 接口定义中没有该方法的定义。

(4) 本程序的其他内容与实例 1 相似，不再赘述。

7.4.4 客户端的 Java 语言程序设计

下面是 CORBA 客户端程序，该程序被保存在 TestCorbaClient.java 文件中。

```
//TestCorbaClient.java
import Test.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class TestCorbaClient
{
    static Istruct IstructImpl;
    public static void main(String args[])
    {
```

```
        try
        {
            ORB orb = ORB.init(args,null);
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            String name = "Istruct";
            IstructImpl = IstructHelper.narrow(ncRef.resolve_str(name));
            Test.IstructPackage.sData sDataRef =
                                    new Test.IstructPackage.sData("Nothing",100);
            IstructImpl.setValue(sDataRef);
            Test.IstructPackage.sDataHolder sHolderRef =
                                        new Test.IstructPackage.sDataHolder( );
            IstructImpl.getValue(sHolderRef);
            Test.IstructPackage.sData sDataRet = sHolderRef.value;
            java.lang.String str = sDataRet.str;
            int len = sDataRet.len;
            System.out.println("Str = " + str);
            System.out.println("Len = " + len);

            String strc = "One World,One Dream";
            IstructImpl.setString(strc);
            StringHolder sHolder = new StringHolder( );
            IstructImpl.getString(sHolder);
            String s = sHolder.value;
            System.out.println("sHolder = " + s);
            IstructImpl.shutdown( );
            try
            {
                IstructImpl.setString("Hello!");
                IstructImpl.getString(sHolder);
            }
            catch(SystemException e)
            {
                System.out.println("Exception: Error occurred!");
                System.exit(1);
            }
            s = sHolder.value;
            System.out.println("sHolder = " + s);

        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

说明：

(1) 本客户端程序通过利用 Holder 类参数来实现从服务器向客户端传递数据的功能。

为了能够接收到 Holder 类参数，必须定义 Holder 类对象并传递给相应的方法。

(2) 程序中 IstructImpl. shutdown( )；语句用于关闭 ORB 对象，同时停止服务器的运行。一旦执行了该语句，则在执行其后面的分布式对象的调用语句时将发生异常。

(3) 本程序的其他内容与实例 1 相似，不再赘述。

7.4.5 Java 类的编译

由前述可知，当前工作目录是 CORBA1，也就是 IDL 接口定义、服务器程序和客户端程序都被保存在 CORBA1 目录下，同时，由于 IDL 接口定义中的模块名是 Test，因此映射后的结果将被保存在 CORBA1\Test\目录下，这样，可采用下述命令对源程序进行编译：

```
C:\CORBA1>javac *.java Test\*.java
C:\CORBA1>
```

7.4.6 启动 orbd

orbd 的启动命令如下：

```
C:\CORBA1>start orbd -ORBInitialPort 1050
```

7.4.7 服务器端程序的执行

orbd 正常启动了之后，就可以启动服务器程序了。另开一个窗口后，在 DOS 命令行上输入如下命令：

```
C:\CORBA1>java TestCorbaServer -ORBInitialPort 1050 -ORBInitialHost localhost
```

将显示如下信息：

```
TestCorbaServer starting…
```

CORBA 服务器程序启动以后，就处于接收调用请求的状态。

7.4.8 客户端程序的执行

CORBA 服务器程序启动了之后，就可以运行客户端程序了。另开一个窗口后，在 DOS 命令行上输入如下命令：

```
C:\CORBA1>java TestCorbaClient -ORBInitialPort 1050 -ORBInitialHost localhost
```

将显示如下执行结果：

```
Str = Nothing
Len = 100
sHolder = One World,One Dream
Exception:Error occurred!
```

如果去掉客户端程序 TestCorbaClient. java 中的 IstructImpl. shutdown( )；语句，则将显示如下执行结果：

```
Str = Nothing
Len = 100
sHolder = One World,One Dream
sHolder = Hello!
```

至此，实例 2 的 CORBA 应用程序开发和运行过程就结束了。

7.5 CORBA 实例 3：Factory 对象的定义与使用

在前面介绍分布式对象引用的获取方法时，曾提到过可以利用 Factory（工厂）对象来获取对象的引用。实际上，在 4.1 节所给出的问题描述中，由于 Control 接口中定义的 openAccount( )方法能够返回实现 Account 接口的对象引用，因此，实现 Control 接口的对象就是 Factory 对象。

本节将详细介绍 Factory 对象的定义与使用，利用本节的方法可以很容易地实现在 4.1 节中所给出的问题。本节的例子既涉及 Factory 对象的定义与使用，同时又涉及在一个模块（module）中定义多个接口（interface）的情况，通过对本节的学习，能够使读者进一步掌握利用 CORBA 来解决实际问题的能力。

回调（callback）技术是在实现分布式对象应用系统时经常使用的一种程序设计技术，本节给出的 Factory 对象的实现方法完全适用于回调技术。有关回调技术的具体内容请参阅后续章节的内容。

7.5.1 问题描述与 IDL 接口定义

本例题主要考察 Factory 对象的定义与使用，也就是如何通过一个分布式方法来返回另一个分布式对象的引用。为了叙述问题方便，对 4.1 节的问题进行了简化，同时，不考虑异常处理问题。下面给出其 IDL 接口定义。

```
//CallbackTest.idl
module CallbackTest
{
    interface Callback
    {
        void setStr(in string str);
        string getStr( );
    };
    interface CallbackRef
    {
        Callback getRef( );
        string getStr(in string str);
    };
};
```

说明：

（1）在 CallbackTest 模块中定义了两个接口，分别是 Callback 和 CallbackRef。每个接口中都声明了两个操作。

(2) 在 CallbackRef 接口中定义的 getRef( )操作是以前没有遇到过的,该操作的主要特点是其返回值是另一个分布式对象的引用。根据 Factory 对象的定义,实现 CallbackRef 接口的分布式对象被称为 Factory 对象。

7.5.2 服务器程序设计

下面是 CORBA 服务器程序,该程序被保存在 CallbackCorbaServer.java 文件中。

```
//CallbackCorbaServer.java
import CallbackTest.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties;

//实现 Callback 接口的分布式对象实现类
class CallbackImpl extends CallbackPOA
{
    String s;
    //构造方法
    public CallbackImpl( )
    {
        s = "";
    }
    //接口中定义的方法的实现
    public void setStr(String str)
    {
        s = str;
        return;
    }
    public String getStr( )
    {
        return s;
    }
}

//实现 CallbackRef 接口的分布式对象实现类
class CallbackRefImpl extends CallbackRefPOA
{
    public static ORB orb;
    public CallbackImpl callbackServant = new CallbackImpl( );
    public static POA rootPoa;
    public Callback callbackRef;
    //构造方法
    public CallbackRefImpl( )
    {
        try
        {
```

```
            //激活所创建的 callbackServant 对象
            rootPoa.activate_object(callbackServant);
            //生成分布式对象引用
            org.omg.CORBA.Object
                    temp = rootPoa.servant_to_reference(callbackServant);
            callbackRef = CallbackHelper.narrow(temp);
        }
        catch(Exception e)
        {
            System.out.println("Servant Already Active!");
            System.exit(1);
        }
    }
    //接口中定义方法的实现
    public Callback getRef( )
    {
        //返回分布式对象引用
        return callbackRef;
    }
    public String getStr(String str)
    {
        return str;
    }
}

//服务器进程类
public class CallbackCorbaServer
{
    public static void main(String args[])
    {
        try
        {
            ORB orb = ORB.init(args,null);
            POA rootpoa =
                 POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager().activate();
            //将 ORB 对象和 rootPoa 对象保存到 CallbackRef 接口的实现类中
            CallbackRefImpl.orb = orb;
            CallbackRefImpl.rootPoa = rootpoa;
            //生成 Factory 对象
            CallbackRefImpl RefImpl = new CallbackRefImpl();
            //获取 Factory 对象的引用
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(RefImpl);
            CallbackRef href = CallbackRefHelper.narrow(ref);
            org.omg.CORBA.Object objRef =
                              orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            //将 Factory 对象引用以“Callback”为名登录到命名服务中
            String name = "Callback";
            NameComponent path[] = ncRef.to_name( name );
            ncRef.rebind(path,href);
```

```
            System.out.println("CallBackCorbaServer starting...");
            orb.run();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

说明：

(1) 为了在 getRef( )方法中返回 CallbackImpl 对象的引用，需要在 CallbackRefImpl 类的构造方法中激活所创建的 callbackServant 对象，并将其引用保存起来，以便需要时返回。

(2) 需要注意的是，getRef( )方法的返回值类型是 Callback，而不是 CallbackImpl。

7.5.3 客户端程序设计

下面是 CORBA 客户端程序，该程序被保存在 CallbackCorbaClient.java 文件中。

```
//CallbackCorbaClient.java
import CallbackTest.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class CallbackCorbaClient
{
    public static void main(String args[])
    {
        try
        {
            ORB orb = ORB.init(args,null);
            org.omg.CORBA.Object objRef =
                    orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            //以 Callback 为名，从命名服务中获取分布式对象的引用
            String name = "Callback";
            CallbackRef rImpl = CallbackRefHelper.narrow(ncRef.resolve_str(name));
            //调用 CallbackRef 接口中定义的方法
            String st = rImpl.getStr("Nothing is hard");
            System.out.println("From CallbackRef:" + st);
            //调用 CallbackRef 接口中定义的方法以获取 Callback 对象引用
            Callback cImpl = rImpl.getRef( );
            //调用 Callback 对象中的方法
            cImpl.setStr("in this world.");
            String s = cImpl.getStr( );
            System.out.println("From Callback:" + s);
        }
        catch (Exception e)
        {
            e.printStackTrace();
```

```
        }
    }
}
```

说明：

(1) 接收 getRef( )方法返回值的变量类型必须是 Callback，而不能是 CallbackImpl。

(2) 从 Factory 对象获取了分布式对象引用之后，就可以利用该对象引用来调用其分布式方法，这同从命名服务获取对象引用是一样的。

7.5.4 语言映射、编译与运行

假如 CalbackTest. idl 文件、CallbackCorbaClient. java 文件和 CallbackCorbaServer. java 文件都被保存在 C:\CORBA2 目录下，则其语言映射、编译与运行过程如下。

1. 语言映射

```
C:\CORBA2>idlj - fall CallbackTest.idl
```

2. 编译

```
C:\CORBA2>javac *.java CallbackTest\*.java
```

3. 运行

(1) 启动 orbd。

```
C:\CORBA2>start orbd -ORBInitialPort 1050
```

(2) 运行服务器程序。

```
C:\CORBA2>java CallbackCorbaServer -ORBInitialPort 1050 -ORBInitialHost localhost
```

将显示如下信息：

```
CallBackCorbaServer starting...
```

(3) 运行客户端程序。

```
C:\CORBA2>java CallbackCorbaClient -ORBInitialPort 1050 -ORBInitialHost localhost
```

将显示如下执行结果：

```
From CallbackRef:Nothing is hard
From Callback:in this world.
```

7.6 CORBA 实例 4：利用文件方式获取分布式对象引用的程序实现过程

在前面介绍的程序中，都是通过命名服务的方式来获取分布式对象引用的，如果在所利用的 CORBA 系统中不提供命名服务功能的话，则上述程序就不能正常运行。同时，正如在

前面的章节中所介绍的那样，如果CORBA客户端是利用Java Applet来实现的话，则也不适合于利用命名服务的方式来获取分布式对象的引用。在本节将介绍利用文件的方式来获取分布式对象的程序实现过程。由于在前面的实例中都详细地介绍了程序的运行过程，本节主要介绍程序功能的实现过程，而对程序的运行过程不再赘述。

7.6.1 IDL接口定义

下面给出了本例子的IDL接口定义。

```
//Hello.idl
module HelloApp
{
  interface Hello
  {
        string sayHello();
        oneway void shutdown();
  };
};
```

其中，sayHello( )操作用于返回字符串“Hello World!”。

7.6.2 服务器程序设计

下面给出了服务器程序的代码。

```
//HelloWorldCorbaServer.java
import HelloApp.*;
import java.io.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties;

//分布式对象实现类
class HelloImpl extends HelloPOA
{
    private ORB orb;
    public void setORB(ORB orb_val)
    {
        orb = orb_val;              //保存ORB对象
    }
    public String sayHello()
    {
        return "Hello World!";
    }
    public void shutdown()
    {
        orb.shutdown(false);        //关闭ORB对象
    }
}
```

```
//服务器进程实现类
public class HelloWorldCorbaServer
{
  public static void main(String args[])
  {
    try
    {
        // ORB 对象的获取
        ORB orb = ORB.init(args,null);
        POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
        rootpoa.the_POAManager().activate();
        HelloImpl helloImpl = new HelloImpl();
        helloImpl.setORB(orb);
        org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
        //将分布式对象引用转化为字符串
        String RefStr = orb.object_to_string(ref);
        //在当前目录下创建 HelloWorldRef.txt 文件
        String filename = "HelloWorldRef.txt";
        FileOutputStream fos = new FileOutputStream(filename);
        //将分布式对象引用写入所创建的文件中
        PrintStream ps = new PrintStream(fos);
        ps.print(RefStr);
        ps.close( );
        System.out.println("HelloWorldCorbaServer starting…");
        orb.run();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
  }
}
```

说明：在该服务器程序中，利用 ORB 的 object_to_string( )方法将分布式对象引用转换为对应的字符串，并将其保存在客户端能够访问的文件中。需要注意的是，由于不使用命名服务来管理分布式对象引用，因此，在程序中就不需要导入与命名服务有关的包(package)。

7.6.3 客户端程序设计

下面给出了服务器程序的代码。

```
import HelloApp.*;
import java.io.*;
import org.omg.CORBA.*;
public class HelloWorldCorbaClient
{
    static Hello helloImpl;
    public static void main(String args[])
    {
```

```
        try
        {
            // ORB 对象的获取
            ORB orb = ORB.init(args,null);
            String filename = "HelloWorldRef.txt";
            FileInputStream fis = new FileInputStream(filename);
            java.io.DataInputStream dis = new java.io.DataInputStream(fis);
            //从文件中读入字符串化的分布式对象引用
            String RefStr = dis.readLine( );
            //将字符串化的分布式对象引用恢复为实际的对象引用
            org.omg.CORBA.Object obj = orb.string_to_object(RefStr);
            helloImpl = HelloHelper.narrow(obj);
            // 调用 sayHello()分布式方法
            String s = helloImpl.sayHello();
            System.out.println("Str = " + s);
            //调用 shutdown( )分布式方法,关闭 ORB 对象
            helloImpl.shutdown( );
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

说明：客户端程序从文件中读取字符串化的对象引用之后，并不能直接使用，而需要利用 ORB 的 string_to_object( )方法将其恢复为实际的对象引用，然后才能进行利用。

7.6.4 语言映射、编译与运行

语言映射、编译及其运行过程与上述程序完全一样，在此不再赘述。当运行客户端程序时，将显示如下信息：

```
Str = Hello World!
```

7.7 简便的程序调试方法

在调试 CORBA 应用程序时经常会出现这样或那样的问题，因此对程序的修改是很频繁的，同时，需要多次编译和运行程序。为了提高程序调试效率，可以将相关命令写入批处理文件中，这样，在需要时只要输入批处理文件名即可。例如，可以将下述运行服务器程序的命令写入名为 server.bat 的文件中，在需要时直接输入 server 并按 Enter 键即可执行：

```
java CallbackCorbaServer -ORBInitialPort 1050 -ORBInitialHost localhost
```

这样，能够缩短命令的输入，提高程序的调试效率。

第8章 Java RMI 技术

RMI(Remote Method Invocation,远程方法调用)提供了 Java 对象之间相互通信的机制。通过利用 RMI 技术,使得运行于某一台机器上的程序可以远程调用另一台机器上的 Java 语言对象。在 RMI 环境中,一般将分布式对象称为远程对象。本章将学习 RMI 技术,并通过实例来说明使用 Java RMI 进行程序设计的过程。本章中的所有程序都是在给定的环境下调试通过的,感兴趣的读者可以按照所给环境来调试和运行本章的程序。

8.1 Java RMI 远程对象调用过程

利用 RMI 开发的应用程序一般是由服务器和客户端两部分组成的,RMI 提供了服务器和客户端之间相互通信的机制。服务器创建远程对象,并等待客户端调用远程对象中的方法;客户端获取服务器上远程对象的引用,并据此调用远程对象中的方法。这里,客户端获取了分布式对象引用之后,通过该引用对分布式对象方法的调用就如同调用本地对象中的方法一样的简单。

8.2 远程对象

为了定义远程对象,必须要对远程接口及其实现类进行描述。

8.2.1 远程接口

远程接口用于定义需要远程调用的方法,换句话说,没有在远程接口中定义的方法是不能进行远程调用的。

远程接口的定义规则如下:

(1) 远程接口必须声明为 public。

(2) 远程接口必须直接或间接地继承 java.rmi.Remote 接口。

(3) 在接口中声明的远程方法必须抛出 java.rmi.RemoteException 异常,该异常是在进行远程方法调用时,发生网络通信或服务器问题时抛出的异常。

(4) 作为参数或返回值传递的远程对象,在对其进行类型定义时,应该声明为远程接口类型,而不是其实现类的类型。

8.2.2 远程接口的实现类

远程接口的实现类即是实现远程对象的类，在远程接口的实现类中也可以定义在远程接口中没有声明的方法，但这样的方法只能进行本地方法调用，而不能进行远程方法调用。

远程接口的实现类的定义规则如下：

(1) 远程接口的实现类必须继承 java. rmi. server. UnicastRemoteObject 类。

(2) 必须实现远程接口。

(3) 在远程接口实现类中定义构造函数，并抛出 java. rmi. RemoteException 异常信息。

(4) 对远程接口中定义的方法进行实现，并抛出 java. rmi. RemoteException 异常信息。

(5) 在 main 函数中，生成远程对象并进行登录，以便客户端能够进行调用。

(6) 创建并安装安全管理器，以便用于保护系统资源不受所下载的代码的破坏。所有使用了 RMI 的程序都必须要安装一个安全管理器，以保证所下载的代码在执行操作时都要通过一系列的安全检查，否则 RMI 不能在远程方法调用中接收参数和返回执行结果等。

(7) 至少将一个远程对象登录到 RMI 远程对象注册器中，以便使远程对象可以由客户端进行访问。

8.2.3 远程对象的生成

远程对象同本地对象一样都是利用 new 运算符来生成的。从远程对象生成并登录到 RMI 注册器开始，服务器就可以接收来自客户端的调用请求了。

8.3 Stub 与 Skeleton

为了使客户端能够利用在服务器方所生成的远程对象，就必须要利用 Stub 类和 Skeleton 类，这同 CORBA 的静态程序设计是相似的。在进行远程方法调用时，通过 Stub 将参数传递给服务器上的远程方法，同时由 Stub 接收来自服务器的返回值并返回给调用者；Skeleton 在服务器方接收由 Stub 传递过来的参数，并提交给远程方法。同时，Skeleton 还需要将远程方法的返回值传递给客户端的 Stub。

Stub 和 Skeleton 是根据远程接口及其实现类的编译结果利用 rmic 命令生成的，在有的 Java 语言版本中可以直接利用 rmic 命令编译服务器程序来生成 Stub 类和 Skeleton 类。

8.4 启动 RMI 注册器

RMI 注册器是客户端获取远程对象引用的手段。RMI 注册器是以名字来管理远程对象的。在网络环境下，客户端通过指定对象的名字就可以获取远程对象的引用。

在 Windows 环境下，执行下述命令就可以启动 RMI 注册器：

```
C:\>rmiregistry
```

通过利用 java. rmi. Naming 类就可以实现对 RMI 注册器的访问。在 Naming 类中，提

供了下列一些访问注册器的方法：

（1）根据指定的名字将远程对象登录到 RMI 注册器中。

```
void bind(String name,Remote obj)
```

（2）解除以给定的名字登录到注册器中的远程对象。

```
void unbind(String name)
```

（3）根据指定的名字将远程对象登录到注册器中。但是，如果在注册器中已经登录了所给定名字的对象，则先解除原有对象的登录，然后重新登录新对象。

```
void rebind(String name,Remote obj)
```

（4）返回以给定的名字登录到注册器中的远程对象的引用。

```
Remote lookup(String name)
```

Naming 类中的方法所使用的名字的定义形式如下：

```
rmi://host 名:port 号/名字
```

其中，“rmi：”可以省略；“host 名”表示被访问的注册器的主机名；“port 号”表示启动注册器的端口号，如果省略“：port 号”部分，则表示使用默认的 1099 端口号。“名字”表示为远程对象所起的名字。

8.5 RMI 程序设计过程

本节以一个简单的程序为例，来介绍 RMI 程序设计过程。在该例子中只实现一个远程方法，用于返回字符串“Hello World”。

8.5.1 远程接口的定义

远程接口的定义是基于 RMI 程序设计的第一步。假设下面远程接口的定义被保存在 HelloWorld.java 文件中。

```
//HelloWorld.java
import java.rmi.*;
interface HelloWorld extends Remote
{
    String sayHelloWorld() throws RemoteException;
}
```

8.5.2 服务器程序的实现

服务器程序就是远程对象，主要用于实现远程接口中的方法。假设下面远程对象的定义被保存在 HelloWorldObj.java 文件中。

```
//HelloWorldObj.java
```

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloWorldObj extends UnicastRemoteObject implements HelloWorld
{
    public static void main(String args[])
    {
        if (System.getSecurityManager() == null)
        {
            //设定 SecurityManager
            System.setSecurityManager(new RMISecurityManager());
        }
        try
        {
            // 生成远程对象
            HelloWorldObj obj = new HelloWorldObj();
            //以"MyObject"为名字将远程对象登录到注册器中
            Naming.rebind("MyObject",obj);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
    //构造方法
    public HelloWorldObj() throws RemoteException
    { }
    //返回"Hello World"的远程方法
    public String sayHelloWorld() throws RemoteException
    {
      return "Hello World";
    }
}
```

8.5.3 客户端程序的实现

客户端程序主要是利用远程方法来实现特定的功能。假设下面的客户端程序定义被保存在 HelloWorldClient.java 文件中。

```
//HelloWorldClient.java
import java.rmi.*;
public class HelloWorldClient
{
    public static void main(String args[])
    {
        HelloWorld obj = null;
        try
        {
```

```
            // 设定 SecurityManager
            System.setSecurityManager(new RMISecurityManager());
            // 获取分布式对象引用(stub类)
            obj = (HelloWorld)Naming.lookup("rmi://localhost/MyObject");
            //调用分布式方法,并显示返回结果
            System.out.println(obj.sayHelloWorld());
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

8.5.4 类文件的编译

本程序是在 Windows 2000 操作系统下使用 J2SE SDK 1.4.0 环境开发的，为了能够正常利用 Java 开发环境，需要对 classpath 和 path 等环境变量进行设置。

假如上述程序都被保存在下述目录下：

```
D:\RMITest\RMIHello
```

1）服务器程序编译

利用 rmic 命令对服务器程序进行编译。其命令格式如下：

```
D:\RMITest\RMIHello>rmic HelloWorldObj
```

编译后将生成下述文件：

```
HelloWorld.class:远程接口
HelloWorldObj.class:远程对象
HelloWorldObj_Stub.class:stub类
HelloWorldObj_Skel.class:skeleton类
```

2）客户端程序编译

利用 javac 命令对客户端程序进行编译。其命令格式如下：

```
D:\RMITest\RMIHello>javac HelloWorldClient.java
```

编译后将生成下述文件：

```
HelloWorldClient.class
```

8.5.5 启动 RMIregistry

RMIregistry 是能够确定分布式对象引用所在位置的简单的命名服务器，也就是注册器。在启动 RMIregistry 之前，一般需要将 CLASSPATH 的值清空。同时，启动 rmiregistry 的目录中不要存在 class 文件。

RMIregistry 的启动命令如下：

```
D:\>set CLASSPATH =
D:\>rmiregistry
```

需要注意的是，正常启动之后，光标处于闪烁状态，屏幕上不显示任何信息。

8.5.6 运行服务器程序

执行服务器程序的目的是将远程对象登录到 RMIregistry 注册器中。为了启动服务器程序，必须在 SecurityManager 中允许连接到 RMIregistry 上。因此，需要定义 java. policy 文件，其内容如下：

```
grant {
    permission java.security.AllPermission;
};
```

在 Java 命令中，利用-D 选项可以指定 java. policy 文件所在的位置和 stub 类所在的 URL。-D 的属性及其意义如下：

- java. security. policy：用于指定写有 Java 安全策略的文件。
- java. rmi. server. codebase：用于指定 stub 类所在目录的 URL，以便装入 stub 类。

在上述基础上，服务器程序的执行命令如下：

```
D:\RMITest\RMIHello>java - Djava.security.policy = java.policy - Djava.rmi.server.codebase =
  file:///C:\RMITest\RMIHello\ HelloWorldObj
```

需要注意的是，上述命令正常执行之后，光标处于闪烁状态，屏幕上不显示任何信息。

8.5.7 运行客户端程序

执行客户端程序的命令如下：

```
D:\RMITest\RMIHello>java - Djava.security.policy = java.policy HelloWorldClient
```

该命令正常执行以后，将在屏幕上显示如下信息：

```
Hello World
```

至此，RMI 程序设计及其运行过程结束。

需要注意的是，上述程序的定义过程主要用于远程对象与客户端程序都在一台计算机上的情况，如果客户端程序与远程对象分别位于不同的计算机上，则需要对客户端程序中的 Naming. lookup( )进行如下修改：

修改前：

```
obj = (HelloWorld)Naming.lookup("rmi://localhost/MyObject");
```

修改后：

```
obj = (HelloWorld)Naming.lookup("rmi://90.0.0.28/MyObject");
```

其中，90.0.0.28 是运行 RMI 注册器的服务器的 IP 地址。另外，为了运行客户端程序，在客户端程序所在的计算机上还必须存在 java. policy 文件、由客户端程序使用的远程接口以及 Stub 类等文件。

8.6 基于回调技术的 RMI 程序设计

对于以 C/S 或 B/S 方式构筑的系统来讲，客户端一般采用查询(Polling)方式来检测服务器的状态变化，这种方式的处理过程是由服务器对象定义返回服务器状态的方法，客户端通过调用这一方法来获取服务器的资源状态。显然，由于查询方式会使客户端频繁地检测服务器的资源状态，这样将浪费大量的网络带宽。与查询方式不同，回调(Callback)技术的应用将解决网络带宽的浪费问题。回调方式的处理过程是由服务器对象定义登录客户端对象的方法，当服务器的资源状态发生变化时，由服务器对象负责通知已登录的客户端对象。

回调技术在分布式对象应用系统设计过程中应用非常广泛，本节的程序就是利用回调技术实现的。

本节将利用 RMI 和回调技术设计一个简单的即时数据传输程序 DataTrans，在 DataTrans 程序中，向 Applet 中输入的信息将被即时发送给所有连接到服务器上的 Applet 并显示出来。服务器与 Applet 之间通过远程方法调用来实现数据的传递。通过对本例的学习，将进一步理解 RMI 的工作方式，同时将学会回调技术的应用过程和在 Applet 中利用 RMI 技术的过程。

DataTrans 程序是由如下几个文件组成的：

(1) 服务器的远程接口(DataTransServer.java)。

(2) 服务器的远程接口的实现类(DataTransServerImpl.java)。

(3) 客户端的远程接口(DataTransClient.java)。

(4) 客户端的远程接口的实现类(DataTransClientImpl.java)。

(5) Applet 程序(DataTransApplet.java)。

(6) 异常定义类(DataTransServerBusyException.java,DataTransClientNotFoundException.java)。

(7) 启动 Applet 所需要的 HTML 文件(DataTrans.html)。

下面将分别给出上述每个文件的具体定义。

8.6.1 服务器的远程接口

服务器的远程接口定义如下：

```
//DataTransServer.java
public interface DataTransServer extends java.rmi.Remote
{
    String login(DataTransClient c)
            throws DataTransServerBusyException,java.rmi.RemoteException;
    void logout(DataTransClient c)
            throws DataTransClientNotFoundException,java.rmi.RemoteException;
    void sendMessage(DataTransClient c,String s)
            throws DataTransClientNotFoundException,java.rmi.RemoteException;
}
```

说明：该接口定义了注册客户方法 login( )、注销客户方法 logout( )和将客户端的文字信息写入服务器并回调显示到所有客户端的 sendMessage( )方法。这里需要注意的是，

作为参数类型的 DataTransClient 也是远程接口类型，主要提供回调显示即时信息的功能。该接口信息被保存在名为 DataTransServer.java 的文件中。

8.6.2 服务器的远程接口的实现类

```
//DataTransServerImpl.java
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.util.*;

public class DataTransServerImpl extends UnicastRemoteObject
                                        implements DataTransServer
{
    ClientHash tbl = new ClientHash();
    public DataTransServerImpl() throws RemoteException
    {
        super();
    }

    public String login(DataTransClient c) throws DataTransServerBusyException,
                                                   RemoteException
    {
        String clientname = null;
        try
        {
            //保存调用该 login( )方法的客户端对象
            tbl.addClient(c,this.getClientHost());
        }
        catch (ServerNotActiveException e)
        {
            tbl.addClient(c,"<unknown>");
        }
        try
        {
            clientname = tbl.getClientName(c);
        }
        catch (DataTransClientNotFoundException e)
        {
            System.out.println("DataTransServerImpl err:" + e.getMessage());
            e.printStackTrace();
        }
        return clientname;
    }

    public void logout(DataTransClient c)
                throws DataTransClientNotFoundException,RemoteException
    {
        tbl.delClient(c);
    }
```

```
    public void sendMessage(DataTransClient c,String s)
                throws DataTransClientNotFoundException,RemoteException
    {
        String cname = tbl.getClientName(c);
        //将信息发送给所有已登录客户端
        tbl.broadcastMessage(cname + ":" + s);
    }

    public static void main(String args[])
    {
        System.setSecurityManager(new RMISecurityManager());
        try
        {
          //生成客户端使用的分布式对象
          DataTransServerImpl obj = new DataTransServerImpl();

          //为了使客户端能够获得分布式对象的引用
          //需要将所生成的分布式对象引用登录到 RMIRegistry 注册器中
          Naming.rebind("DataTransServer",obj);

          System.out.println("DataTransServer bound in registry");
        }
        catch (Exception e)
        {
          System.out.println("DataTransServerImpl err:" + e.getMessage());
          e.printStackTrace();
        }
    }
}

class ClientHash
{
    final static int CLIENT_MAX = 6;
    int client_num = 0;
    int client_count = 0;

    class Client
    {
        String name;
        DataTransClient robj;
        public Client(DataTransClient r,String s)
        {
            robj = r;
            name = s;
        }
    }
    Hashtable c_hash = new Hashtable();

    public synchronized int addClient(DataTransClient r,String s)
                                    throws DataTransServerBusyException
```

```
{
    int no = client_num;
    if( no >= CLIENT_MAX )
    {
      throw new DataTransServerBusyException("DataTransServerImpl:
                                        DataTransServer is busy!");
    }
    String cname = new String(s + "." + String.valueOf(client_count++));
    c_hash.put(r,cname);
    client_num++;
    //向各注册客户端发送注册信息
    broadcastMessage("<" + cname + " login>");
    return no;
}

public synchronized void delClient(DataTransClient r)
                        throws DataTransClientNotFoundException
{
    String cname = getClientName(r);
    if( cname == null )
    {
      throw new DataTransClientNotFoundException("DataTransServerImpl:
                                            No this client!");
    }
    client_num--;
    c_hash.remove(r);
    //向各注册客户端发送注销信息
    broadcastMessage("<" + cname + " logout>");
}

public synchronized String getClientName(DataTransClient r)
                        throws DataTransClientNotFoundException
{
    String cname = (String)c_hash.get(r);
    if( cname == null )
    {
      throw new DataTransClientNotFoundException("DataTransServerImpl:
                                                No this client!!");
    }
    return cname;
}

public synchronized void broadcastMessage(String s)
{
    for(Enumeration enum = c_hash.keys(); enum.hasMoreElements();)
    {
     try
     {
        DataTransClient c = (DataTransClient)(enum.nextElement());

        //利用回调技术向客户端发送信息
```

```
                c.sendMessage(s);
            }
            catch (Exception e)
            {
                System.out.println("ClientHash exception:" + e.getMessage());
            }
        }
    }
}
```

说明：DataTransServerImpl 类是 DataTransServer 远程接口的实现类，除了实现 DataTransServer 接口中定义的方法之外，还需要实现利用命名服务来注册服务器对象的 main( )方法。

login( )方法用于注册客户，它是每个客户端的 Applet 程序必须第一个调用的服务器对象中的方法，其功能是将客户端传递过来的 DataTransClientImpl 客户端对象登录到 Hashtable 对象中；logout( )方法用于注销客户，也就是清除登录在 Hashtable 对象中的客户端对象；sendMessage( )方法接收来自客户端的即时数据信息，同时依次取出登录在 Hashtable 对象中的客户端对象，并利用客户端对象中的 sendMessage( )方法将即时数据信息输出到每个客户端的画面上。实际上，这一处理过程就是利用回调技术实现的。

8.6.3 客户端的远程接口

```
//DataTransClient.java

public interface DataTransClient extends java.rmi.Remote
{
    void sendMessage(String s) throws java.rmi.RemoteException;
}
```

说明：该接口定义了向客户端画面上输出即时数据信息的 sengMessage( )方法。

8.6.4 客户端的远程接口的实现类

```
//DataTransClientImpl.java

import java.awt.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class DataTransClientImpl extends UnicastRemoteObject
                                 implements DataTransClient
{
    TextArea textarea;
    public DataTransClientImpl(TextArea t) throws RemoteException
    {
        super();
        textarea = t;
    }
    public void sendMessage(String s) throws RemoteException
```

```
    {
        textarea.append(s + "\n");
    }
}
```

说明：DataTransClientImpl 类是 DataTransClient 接口的实现类，其中定义的 sendMessage( )方法主要用于回调显示所要发送的数据，也就是服务器接收到来自客户端的数据后，通过调用每个客户端的 sendMessage( )方法，将数据显示到每个客户端的画面上。

8.6.5 异常类的定义

下面的两个类用于定义在远程接口定义中所使用的异常类，其定义方式需严格按照 Java 语言中的异常类的定义规则来进行。

```
// DataTransServerBusyException.java
public class DataTransServerBusyException extends Exception
{
    public DataTransServerBusyException(String s)
    {
        super(s);
    }
}
// DataTransClientNotFoundException.java
public class DataTransClientNotFoundException extends Exception
{
    public DataTransClientNotFoundException(String s)
    {
        super(s);
    }
}
```

8.6.6 Applet 程序与 HTML 文件的定义

```
//DataTransApplet.java
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class DataTransApplet extends java.applet.Applet
                             implements ActionListener
{
    Label title;
    TextArea display;
    TextField input;
    DataTransClientImpl client;
    DataTransServer server;
```

```
    String clientname;
    boolean init = false;

    public void init()
    {
    title = new Label("",Label.CENTER);
    display = new TextArea(20,40);
    display.setEditable(false);
    input = new TextField(40);
    input.addActionListener(this);
    setLayout(new GridLayout(3,0));
    add(title);
    add(display);
    add(input);
    try
    {
      //获得服务器的远程对象引用
      server = (DataTransServer)Naming.lookup("rmi://" +
                        getCodeBase().getHost() + "/DataTransServer");
      // 生成服务器使用的客户端远程对象
      client = new DataTransClientImpl(display);
      //将客户端远程对象登录到服务器中
      clientname = server.login(client);
      title.setText(clientname);
      init = true;
    }
    catch (Exception e)
    {
      System.out.println("DataTransApplet.init exception:" + e.getMessage());
      title.setText("Server busy!!");
      remove(input);
      remove(display);
    }
}

public void destroy()
{
    try
    {
      if( init )
      {
          server.logout(client);
      }
    }
    catch (Exception e)
    {
        System.out.println("DataTransApplet exception:" + e.getMessage());
        e.printStackTrace();
    }
    removeAll();
}
```

```
public void actionPerformed(ActionEvent ev)
{
        if( ev.getSource() == input )
        {
          if( ev.getID() == ActionEvent.ACTION_PERFORMED )
          {
              try
              {
                  //将输入信息发送给服务器
                  server.sendMessage(client,input.getText());
              }
              catch (Exception e)
              {
                  System.out.println("DataTransApplet.actionPerformed exception:" +
                                                                  e.getMessage());
              }
              input.setText(""); //清空已发送的信息
          }
      }
    }
}
```

说明：本程序是 Applet 程序，需要按照 Applet 程序的结构来定义。

由于 Applet 程序需要嵌入在 HTML 文档中使用，因此，下面给出 HTML 程序的定义。该 HTML 程序被保存在 DataTrans.html 文件中。

```
<HTML>
<HEAD>
<TITLE>DataTrans</TITLE>
</HEAD>

<BODY>
<h1>DataTrans APPLET</h1>
<hr>
<p>
<center>
<APPLET CODE = "DataTransApplet"
    WIDTH = 300 HEIGHT = 200>
<hr>
Your browser can't understand APPLET tag.
<hr>
</APPLET>
</center>
</BODY>

</HTML>
```

8.6.7 定义 java.policy 文件

java.policy 文件的具体内容如下：

```
grant {
    permission java.security.AllPermission;
};
```

8.6.8 编译与运行

假如上述程序被保存在 D:\RMITest\RMIDataTrans 目录下，则其编译与运行过程如下：

(1) 服务器程序编译。

```
D:\RMITest\RMIDataTrans>rmic DataTransServerImpl DataTransClientImpl
```

(2) Applet 程序编译。

```
D:\RMITest\RMIDataTrans>javac DataTransApplet.java
```

(3) 启动 RMIRegisty。

```
D:\>set classpath=
D:\>rmiregistry
```

(4) 另开一个窗口，运行服务器程序。

```
D:\RMITest\RMIDataTrans>java -Djava.security.policy=java.policy
-Djava.rmi.server.codebase=file:///C:\RMITest\RMIDataTrans\ DataTransServerImpl
```

(5) 另开一个窗口，运行 Applet 程序。

```
D:\RMITest\RMIDataTrans>appletviewer DataTrans.html
```

(6) 再另开一个窗口，运行 Applet 程序。

```
D:\RMITest\RMIDataTrans>appletviewer DataTrans.html
```

这样，就可以在两个 Applet 程序之间进行信息交流了。在一个 Applet 的输入框中输入完信息并按 Enter 键后，该信息会即时显示在其他 Applet 的画面上。

在本节中，利用 Java RMI 及其回调技术实现了数据的即时传输程序，从中可以看出回调技术在分布式对象系统设计过程中的重要性。感兴趣的读者也可以将该程序利用 CORBA 技术进行改写，从中可以进一步了解 Java RMI 与 CORBA 的异同。

参考文献

[1] 小野沢博文. CORBA 完全解説. ソフト・リサーチ・センター. 2000
[2] 松野良藏. Java＋CORBA 分散オブジエクト构筑. 翔泳社. 1999

- 教学目标明确，注重理论与实践的结合
- 教学方法灵活，培养学生自主学习的能力
- 教学内容先进，反映了计算机学科的最新发展
- 教学模式完善，提供配套的教学资源解决方案
- 可下载教学资料：http://www.tup.tsinghua.edu.cn

ISBN 978-7-302-38596-7
9 787302 385967 >
定价：35.00元